Utilizing hardware features for secure thread managemen

Isa, Haruna R.

Monterey, California. Naval Postgraduate School

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

UTILIZING HARDWARE FEATURES FOR SECURE THREAD MANAGEMENT

by
Haruna R. Isa

December 1998

Thesis Advisor: Cynthia Irvine
Second Reader: William Shockley

Approved for public release; distribution is unlimited

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1998	Master's Thesis

4. TITLE AND SUBTITLE
UTILIZING HARDWARE FEATURES FOR SECURE THREAD MANAGEMENT

5. FUNDING NUMBERS

6. AUTHOR(S)
Isa, Haruna R.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(maximum 200 words)*

Transaction processing (TP) applications are of use when solving a wide variety of data processing problems. Current, commercial TP systems do not possess the ability to manage information at multiple security levels with high assurance. Department of Defense and Department of Navy Command, Control, Communication, Computers and Intelligence (C4I) applications handle information over a wide variety of classifications and compartments. The existence of TP applications that can securely process information of different classifications (with assurance) would save the DoD the need to create separate single level systems to process all necessary information. A trusted computing base (TCB) and security kernel architecture for supporting multi-threaded, queue-driven transaction processing applications in a multilevel secure environment has been designed. Intel's Pentium CPU architecture provides hardware with two distinct descriptor tables. One is used in the usual way for process isolation while the other is used for thread isolation. This allocation, together with an appropriately designed scheduling policy, permits us to avoid the full cost of process creation when only switching between threads of different security classes in the same process. Where large numbers of transactions are encountered on transaction queues, this approach has benefits over traditional multilevel systems.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Transaction Processing, Multilevel Secure Operating System, Thread Management, Intel Pentium Microprocessor	157
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFI- CATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-1

Approved for public release; distribution is unlimited

UTILIZING HARDWARE FEATURES FOR SECURE THREAD MANAGEMENT

Haruna R. Isa
Lieutenant, United States Navy
B.S., University of Pittsburgh 1991

Submitted in partial fulfillment of the
requirements for the degrees of

MASTER OF SCIENCE IN SYSTEMS TECHNOLOGY (SCIENTIFIC AND TECHNICAL INTELLIGENCE)

AND

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL

December 1998

Author [redacted]
Haruna R. Isa

Approved by: [redacted]
Cynthia E. Irvine, Thesis Advisor

[redacted]
William R. Shockley, Second Reader

[redacted]
Dan Boger, Chairman, Department of Computer Science and Chairman, Department of Command, Control, Communications, Computers and Intelligence

ABSTRACT

Transaction processing (TP) applications are of use when solving a wide variety of data processing problems. Current, commercial TP systems do not possess the ability to manage information at multiple security levels with high assurance. Department of Defense and Department of Navy Command, Control, Communication, Computers and Intelligence (C4I) applications handle information over a wide variety of classifications and compartments. The existence of TP applications that can securely process information of different classifications (with assurance) would save the Department of Defense the need to create separate, single level systems to process all necessary information.

A trusted computing base (TCB) and security kernel architecture for supporting multi-threaded, queue-driven transaction processing applications in a multilevel secure environment has been designed. Intel's Pentium CPU architecture provides hardware with two distinct descriptor tables. One is used in the usual way for process isolation while the other is used for thread isolation. This allocation, together with an appropriately designed scheduling policy, permits us to avoid the full cost of process creation when only switching between threads of different security classes in the same process. Where large numbers of transactions are encountered on transaction queues, this approach has benefits over traditional multilevel systems.

TABLE OF CONTENTS

LIST OF FIGURES

I. INTRODUCTION

Although many of the traditional transaction processing (TP) systems fulfill a variety of commercial needs, there are applications that lend themselves to TP but levy the added requirement to provide multilevel security (MLS). Examples might be a military command and control (C^2) system supporting users at different access classes or a central financial clearinghouse simultaneously processing transactions for several competing banks. In our notional C^2 system, this single TP system receives input in the form of positional data from a variety of sources. Although the format of this positional data might be the same for different input sources, the classification of the data varies based upon the sensitivity of the source.

As an example, the positional data for a ship which itself broadcasts its location might be CONFIDENTIAL (C) whereas the same positional data obtained through some sensitive intelligence sensor might be classified TOP SECRET (TS). Scenarios such as this, where information of the same type has varying classifications based upon source, are commonplace in military intelligence and command and control applications. We can also see the need for such a system to support users of varying clearances as well. For instance, the commander of a small ship might be cleared to view data up to and including SECRET (S) whereas the commander of a battle-group might be cleared to view data up to and including TS. In a typical C^2 system, the output viewed by the users might take the form of a graphical display showing the position of various units. Although these various displays might be driven by a shared MLS database, updating this database is the responsibility of the TP system.

This is a classic application for TP systems: small packets or input (positional reports) are processed by programs (TP tasks) to produce output (updating a shared MLS database.) Traditional high assurance multilevel operating systems are designed to support general purpose processing. Users log in and run a series of arbitrary processes that are dynamically created and terminated. The scheduler, similar to that in a

traditional multiprocessing system (Stallings, 1998), is organized to support users at different security levels running a variety of tasks.

In contrast, transaction processing systems, once initialized, are more like an assembly line or Petri net. The tasks are static and units of work (transactions) flow from one task to the next. Transactions are placed on input queues. A transaction processing task will work on items from its input queues until the queues are empty and will place work on the queues for the next task in the work flow. Rescheduling does not take place until the task has nothing to do (Bernstein and Newcomer, 1997). (Note that for now we consider queues to be unbounded and do not consider rescheduling caused by encountering a limit on queue size.) In a traditional high assurance multilevel operating system, a process switch is required when changing from one security level to another.

To process transactions on a general purpose high assurance multilevel operating system, process switches would be required each time an item at a different security level was encountered on the input queue. This would result in a severe performance penalty even though the same task is being performed on each transaction.

In this thesis, an architecture is presented for a high assurance multilevel security system specifically intended to support TP. Chapter II presents some background on the problem, the scope of this project and discusses the architecture of a typical commercial TP system. Chapter III discusses some alternate ways to create an MLS TP system. These alternatives include creating separate, single-level TP systems for each possible classification/compartment pair or creating a TP system on top of a current MLS general purpose system. Both alternatives are argued to be inadequate. Chapter IV briefly introduces the security features of the Intel Pentium microprocessor that will be utilized. Chapter V presents the architecture and provides an overview of the modules that comprise it. Chapters VI through VIII present the simplified specification for the various components of the system. Chapter IX concludes the thesis.

II. BACKGROUND

A. PROBLEM STATEMENT

The primary reason TP systems are developed independently of general purpose operating systems is performance. Either the response of the system or the throughput must be optimized (Bernstein and Newcomer, 1997). This creates a problem when dealing with multilevel transactions. In order to maintain assurance against compromise, MLS systems must often perform elaborate steps when switching between processes with differing classifications. These steps are to ensure that the new process cannot access anything that was used by the old process, in accordance with a chosen security policy. These steps also require processor time that cannot be used to process transactions, creating inefficiency.

B. SCOPE

The Intel Pentium series of microprocessors implement some interesting security features in hardware. These features (two general purpose descriptor tables and four privilege levels) allow efficient switching between processes while still providing assured isolation between the processes.

This thesis presents a preliminary, general architecture and simplified specification for an MLS TP system that exploits these microprocessor features. It does not present a complete, implemented system nor a detailed specification appropriate for formal assurance arguments. The simplified specification is for the modules, functions, databases and interfaces that would comprise the completed system. This specification should be viewed as scaffolding upon which future implementation work can be based and is subject to change.

C. TRADITIONAL TRANSACTION PROCESSING

Although the architecture of a traditional transaction process system has been briefly touched upon, a formal introduction is relevant to our architecture. A transaction is defined as a unit of input (Bernstein and Newcomer, 1997). Its lifetime runs from when it is initially enqueued to the system to when it is finally dequeued and leaves the system or is destroyed. We further define a TP task as the actual TP program executed to process a single transaction (unit of input). The transactions are removed from the queues and processed by the TP tasks. TP tasks can also add new transactions, or return transaction previously removed, to the queues. The nature of the TP tasks or the transactions can vary, but the overall structure remains the same. Normally, the function performed by a TP task will involve the update of some shared database, but it could just as easily result in the generation of a new transaction containing a single value.

1. Basic TP Architecture

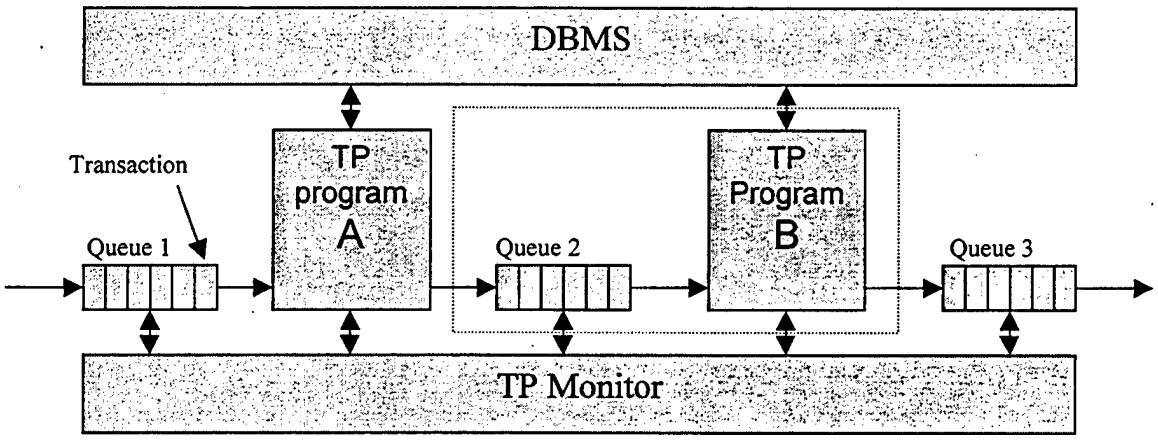

Figure 2.1: Traditional Transaction Processing Architecture

In a typical queued TP system (such as Figure 2.1), queues are used as the buffer for input and output from the TP tasks. A TP task is run by a TP manager to process a transaction or multiple transactions from one or more input queues. Multiple TP tasks might be run (in a multiprocessed environment), all taking transactions from input

queues. The TP tasks write their results (if any) as new transactions which are enqueued on output queues.

The TP tasks access the transactions by performing operations on the queues. The **dequeue** operation is used to get a transaction from a queue. The transaction is not immediately deleted from the queue but instead is saved in a holding area pending notification from the task that the transaction has committed. Once the task has completed processing the transaction, it may perform an **enqueue** operation on another queue to pass along its results in the form of a new transaction. Finally, a **commit** operation indicates that the results of processing the transaction have been durably recorded and the transaction can be permanently deleted from the holding area of the input queue.

A TP system might consist of several queues with different types of TP tasks being run to process the inputs. The system might consist of multiple copies of a single TP task (if it only performs one function) or several hundred different types of TP tasks to process a wide variety of input types. Additionally, there could be a single input queue which holds all inputs of all types or perhaps several different input queues each holding a specific type of input. The same might apply to the output queues. Again, the central working of the TP system remains unchanged: 1) a unit of input is received and placed on an input queue, 2) a TP task is run to process a single unit or multiple units of input (from queues), and 3) the TP task completes and enqueues results (in the form of new transactions) on output queues or updates some shared database. This is the paradigm we will use for our design.

TP systems also usually implement logging. Logging of transactions is necessary for graceful recovery with a minimum loss of transactions. Should the system fail while several TP tasks are in the midst of processing transactions, the transaction that were being processed must not be lost when the system recovers. Although this is a requirement for implemented TP systems, it is not included in the architecture presented, but would need to be part of any final implementation.

TP systems also need to allow variable workflow. For example, consider the set of TP tasks and queues in Figure 2.2. After TP task B has completed processing a transaction it may enqueue a new transaction on both queues for TP tasks C and D. The TP system's queue management needs to ensure such dynamic workflow is allowed. The system becomes much more complex when we consider that the TP tasks might be at different sensitivity levels. A sensitivity level is a reflection of the damage that would result should the information be divulged to parties who are not cleared to receive it. These sensitivity levels are typically part of a hierarchical policy, such as the Bell and LaPadula model (Bell and LaPadula, 1976), so a task with a high sensitivity level can read all information at its level and below but is prohibited from writing information to a lower level. Sensitivity levels are analogous to military classifications. In Figure 2.2, if TP tasks A and C were at a high sensitivity level and TP tasks B and D were at a low sensitivity level, then TP task A would not be allowed to enqueue (write) a new transaction for TP task D (this would be a violation of the security policy.)

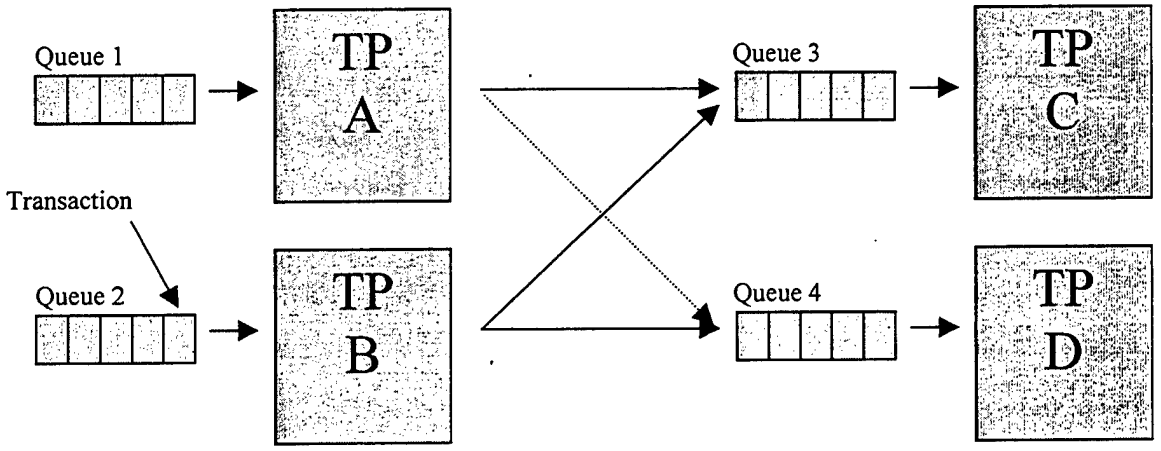

Figure 2.2: Variable Workflow

2. TP Requirements

There are also four critical properties normally associated with transactions (and which must be maintained by the TP system). These properties are atomicity, consistency, isolation and durability (ACID) (Bernstein and Newcomer, 1997).

A transaction is atomic in the sense that it either completes fully or does nothing at all, there is no notion of partial work. The TP system must have a mechanism in place to ensure that the results of the TP task do not take effect until the task successfully completes. A successful completion of a transaction is called *commit* and the failure of a transaction is called *abort*.

Consistency in TP applications normally applies to any shared database accessed by multiple tasks. A database that is consistent prior to the running of a transaction should be consistent following completion of the transaction whether successful or not. Database consistency normally refers to entity and referential integrity and possibly the maintenance of some invariant.

The isolation of a TP application refers to the serializability of transactions. The transaction runs as if it were running alone with no other transactions. That is, the transaction should not be affected by the concurrent running of any other transaction or any other system activity that might occur during its execution.

Durability means the results of a successful transaction are permanently stored on a medium that will survive the failure of the TP system. Typically this refers to writing the results of processing the transaction onto some type of disk storage medium (audit log).

Additionally, when a transaction updates data on several distributed systems or in a client/server environment, the two-phase commit protocol might be used to ensure successful completion of the transaction updates on both systems (Bernstein and Newcomer, 1997). The details of the two-phase commit protocol are not addressed here and are not central to our design.

The final note we will make about TP systems is that performance is an issue. Typically, a system will be designed for maximum throughput, that is, designed to complete the maximum number of transactions possible in a given amount of time. This is the main reason TP systems are designed as special purpose systems as opposed to modifying some other type of system to support TP. For example, a typical interactive timesharing system could be modified to support TP, but since it was not optimized for such an application, throughput is likely to suffer compared to a system designed exclusively for TP. The main reason for this inefficiency is the overhead normally associated with process switches in typical interactive timesharing systems (Stallings, 1998). A process switch will typically involve the saving of all registers and the switch of all descriptor tables. On the Intel Pentium series of microprocessors switching the Global Descriptor Table (GDT) is a time consuming process. Switching between two TP tasks might not require this full context switch and thus might be accomplished much faster with a smaller context switch.

3. MLS Requirements

The MLS TP architecture presented imposes additional requirements not found in typical TP systems. The system should enforce a mandatory security policy providing MLS. This could be a mandatory confidentiality policy (Bell and LaPadula, 1976), a mandatory integrity policy (Biba, 1977), or both. When a transaction of a given sensitivity level is executing, the system should ensure it can read and write objects only in accordance with the mandatory policy.

Performance is an explicit requirement for the system as well. It should be capable of switching between sensitivity levels as rapidly as possible, consistent with properly enforcing the policy.

III. ALTERNATIVE SOLUTIONS

Alternate approaches to creating an MLS TP system are examined and rejected. The first is the approach traditionally taken; to create wholly separate subsystems which process information at a single level for users cleared to that level. The second approach uses already developed MLS timesharing systems to implement MLS TP.

A. SEPARATE SUBSYSTEMS

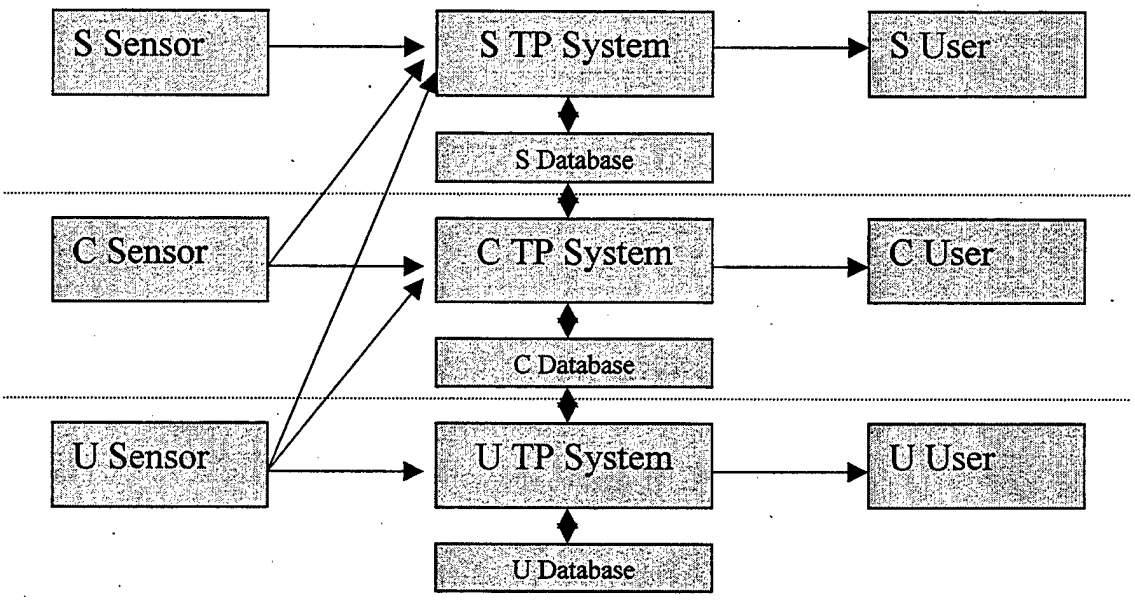

Figure 3.1: A MLS TP System Composed of Separate Single Level TP Systems

The approach taken by most designers to date has been to create almost completely separate systems, each of which processes information of a single level for users cleared to that level as depicted in Figure 3.1. Each subsystem then operates in dedicated mode (DoD, 1985) without the need for any elaborate internal security features.

An MLS TP system is created by building several separate TP systems, each TP system processes information of a specific sensitivity level. Using the initial C^2 example, one system would be created for every level of user that must be supported. For example, if users cleared to CONFIDENTIAL (C), SECRET (S) and TOP SECRET (TS) must be

supported, three separate TP systems must be created. Each system would have its own database. Sensor information at lower levels would be forwarded to higher systems to ensure high level users received all information they are cleared for. For example, a C report would be input, not only to the C TP system, but also to the S and TS TP systems. Thus a TS user would see C, S and TS positional data (as he should).

The advantage of such a TP system of systems is clear. Since each TP system is operating in dedicated mode; there is no need for security mechanisms within each system. As such, commercial of-the-shelf (COTS) TP and database management systems (DBMS) could be used. A high assurance system, such as the Naval Research Laboratory (NRL) pump (Kang, Froscher, and Moskowitz, 1997), is required to push information from the lower sensitivity levels to higher ones. These products have already been developed and tested and are readily available. The fact that most of these products do not provide any security assurance is of little concern since all users who will access them are cleared to see all information contained within them.

The disadvantage of such a system lies in its inefficiency. For example, looking at Figure 3.1, U data is processed identically three separate times by the systems at each level. This needlessly wastes processing power. Another serious drawback to such a system is the triple storage requirements. Assuming U data is most prevalent in the system, the requirement to store the U data and results in three separate places is a significant storage burden.

Another disadvantage to such a system lies in its inability to process a wide variety of data. If we have the need to separate not just classifications but also several compartments, we can see that the number of separate systems needed quickly grows. When dealing with information and users cleared to several classifications with several compartments, the number of needed systems to ensure separation becomes prohibitive. Also, since all the systems are separate, the TP goal of consistency might be difficult to achieve. Since the C system cannot control when the S system performs its updates (and

vice-versa), a C user and an S user, both viewing C data, might not be looking at the same thing.

Finally, users at high sensitivity levels are unable to distinguish between low and high information. They must treat all information as high since, with no underlying assured mechanism to associated labels with information, no labels can be trusted. Downgrading would be difficult, if not impossible. This same problem makes it possible for low information to foul the high system. For instance, in intelligence systems, high information might be implicitly considered more reliable due to the sources. When low and high information are mixed together in a single system with untrusted or no labels, such reliability judgements are no longer possible.

Although such a system leverages COTS products to solve the problem, it is unnecessarily inefficient and fails when required to handle a wide variety of data and user classifications and compartments.

B. MLS TIMESHARING SYSTEM

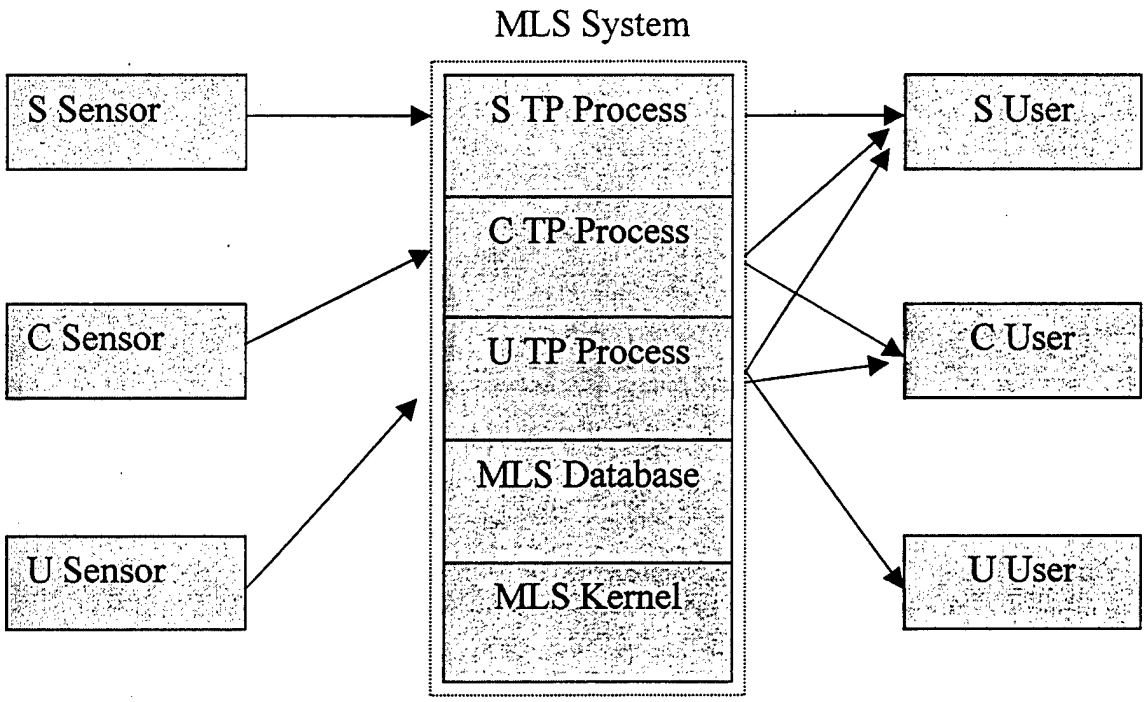

Figure 3.2: An MLS TP System Implemented On Top of a General Purpose MLS System

The second alternative considered, but rejected, is placing a TP system on top of an already developed and accredited MLS timesharing system (depicted in Figure 3.2). In such a system, a single TP task would run as a separate process. The system would support TP tasks of differing levels by creating the required number of processes. Using the system pictured in Figure 3.2 as an example, when a C input is received the system would switch to the C process, within which the C TP task would perform its function. If an S input were received next, the system would switch from the C process to the S process, within which the S TP task would perform its functions. Such switches from process to process would continue to handle the stream of incoming transactions.

This scheme has advantages not found in the scheme using separate systems. First, we again leverage already developed products (the trusted MLS timesharing systems). Additionally, since all processing is done on a single system we avoid the

inefficiencies and waste of using multiple, separate systems. We also do not have a problem handling a wide variety of data and user access classes (classification and compartment) since it only means the creation of additional processes, not additional separate systems. We also gain consistency using this alternative. Since all users are viewing the same U data, there can be no inconsistency between and S and U user's view of U data. Additionally, using this scheme, we have labels that can be trusted.

The primary reason we reject this scheme is that the MLS timesharing systems are not designed for TP. The switches between processes of differing levels might be invoked often. Since TP processes of different sensitivity levels execute in different processes, switches between processes occur whenever transactions of different sensitivity levels must be processed. Switches between processes in a general purpose MLS system requires at least as much time as process switches in single-level general purpose systems, often more. Access level context switches are costly in time when performed on an MLS timesharing system. We did not encounter this problem when using separate systems since each system runs in dedicated mode without worrying about security, there is no need for any context switches. So, although we have removed the unnecessary redundancy we suffer a significant, unacceptable performance penalty when moving between TP tasks of differing access classes.

IV. PROCESSOR INTRODUCTION

The MLS TP architecture presented is targeted toward the Intel Pentium series of microprocessors. This series of processors implement descriptor based segmented memory and multiple privilege levels. These architectural features provide a good foundation for an MLS operating system.

The Pentium series microprocessors provide two modes of operation; real and protected mode. Real mode is provided for backward compatibility and does not provide any of the memory protection required for a multitasking system, much less for an MLS operating systems. Protected mode, however, provides hardware enforcement of memory accesses based upon privilege levels, available descriptors and descriptor attributes. Protected mode is the target of the architecture.

A. PRIVILEGE LEVEL STRUCTURE

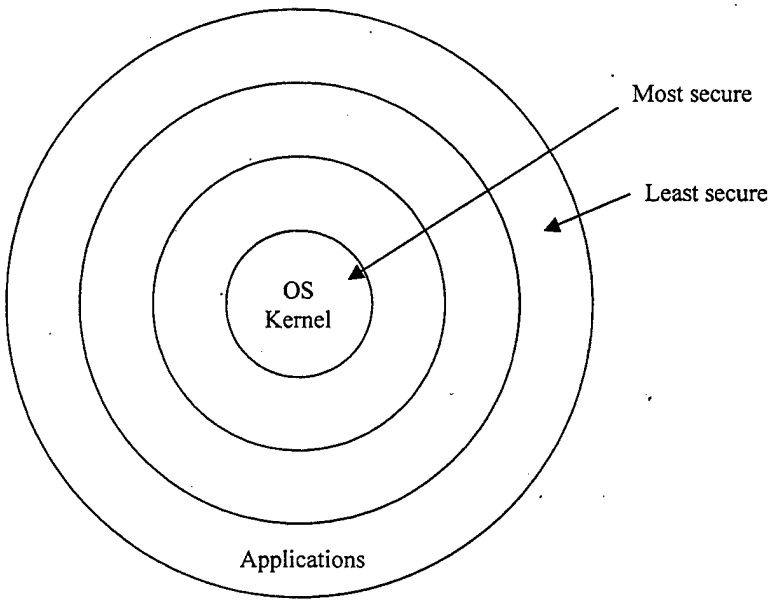

Figure 4.1: Pentium Privilege Level Structure

The Intel microprocessors use a four privilege level structure (Figure 4.1) numbered 0 through 3 (Intel, 1997). Privilege level 0 is the most privileged and privilege

level 3 is the least privileged. As depicted in Figure 4.2, a process running in a given privilege level has access to segments with descriptors at its level and above (less privileged.)

Access to any function in a more privileged level is only allowed through a gate (Figure 4.3). The gate provides entry points to the more privileged levels. The gate thus ensures that access to privileged functions by non-privileged subjects is strictly controlled.

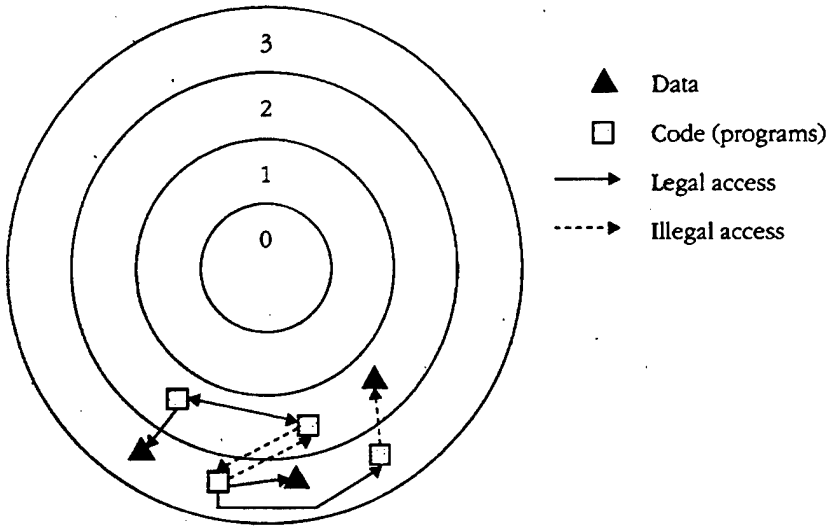

Figure 4.2: Access Between Rings

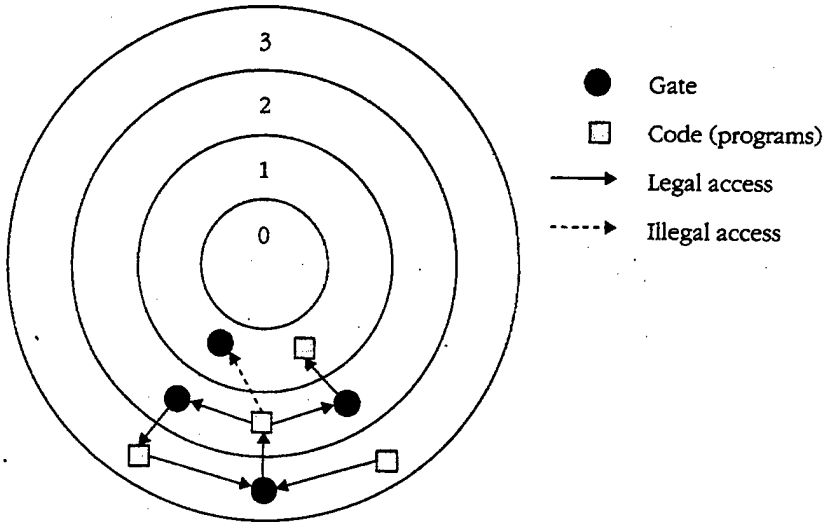

Figure 4.3: Call Paths Through Gates

Through the use of gates, a few functions can be carefully exported from more privileged to less privileged levels. The gates can be designed to call routines that perform complex validation of arguments to ensure that untrusted tasks are not trying to pass invalid arguments to a trusted routine.

B. DESCRIPTOR TABLES

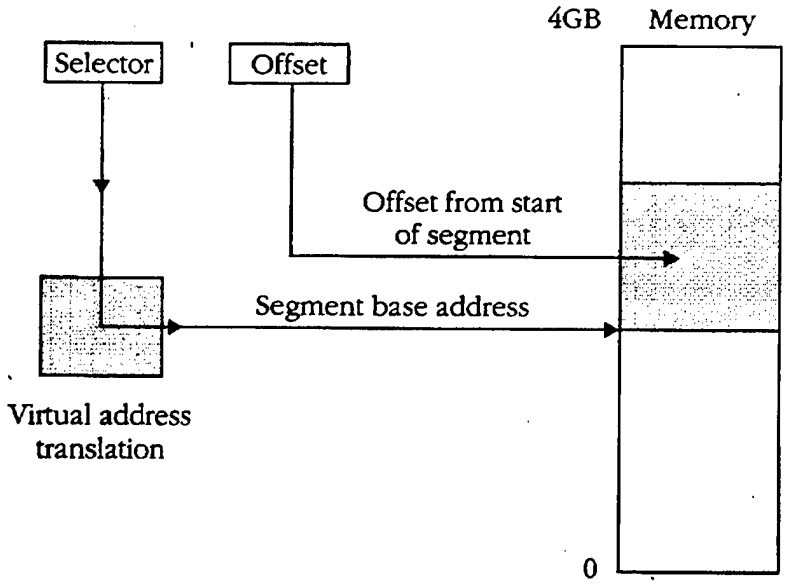

Figure 4.4: Address Translation

The Intel Pentium microprocessor architecture also provides two general-purpose descriptor tables. A global descriptor table (GDT) and a local descriptor table (LDT). As seen in Figure 4.4, any access to memory must be via a selector that references an entry in either the GDT or LDT. The entry in the table would be the descriptor, which provides a description of the segment, including its physical address in memory. The physical base address of the segment in memory is used with the offset to find the linear address referenced. The descriptors have privilege levels associated with them as well, this allows the processor to disallow attempts by less privileged subjects to access more privileged segments. Thus, for a process to access memory, the descriptor describing the memory segment must be loaded into a descriptor table (either the LDT or GDT) and the descriptor must be of the same or lesser privileged level. This access check is performed in hardware making the protection mechanism very efficient.

V. ARCHITECTURE

The goal of this work is to design a TP system which provides the protection of using an MLS timesharing system while avoiding the heavy performance penalty imposed by such systems. Realizing that a TP system might be required to process a variety of transaction types, each of which might have a variety of access classes, a three-tier architecture has been devised, pictured in Figure 5.1.

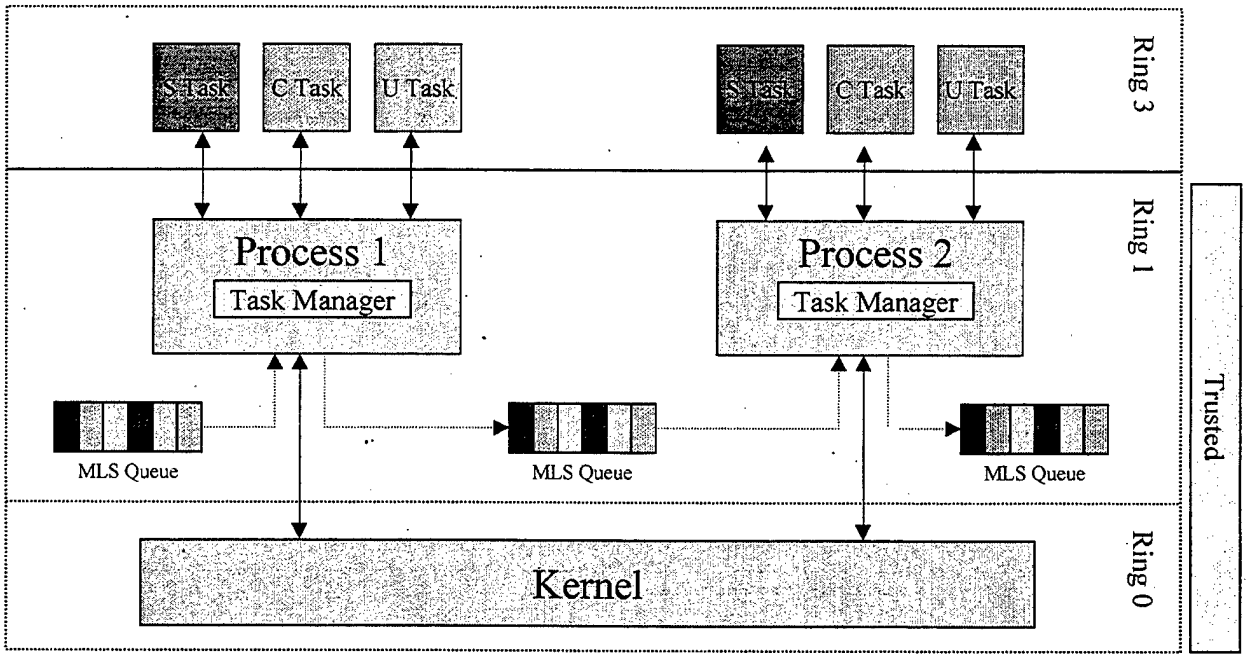

Figure 5.1: MLS TP Architecture

At the core of the system is the security kernel. The kernel is primarily responsible for all memory management functions, all process management and providing eventcounts for process synchronization.

The middle tier of the system contains the multilevel queue manager and the task manager. The multilevel queue manager holds the transactions. The MLS queue manager can handle several queues through which the transactions flow while moving through the system. The queue manager supports operations to create and delete MLS queues as well as to enqueue, dequeue and get work from an MLS queue.

A task manager runs within each process and controls the various single level TP tasks. These TP tasks are the elements within the system that actually perform all processing of the transactions. The task manager will schedule TP tasks, mediate memory management requests to the kernel and mediate queue requests.
Incoming transactions of a given type would be enqueued on the appropriate MLS queue by some trusted input processor. The tasks are the outer layer of the system and are single level entities which process a given transaction type of a given classification. The task manager will get work from the appropriate MLS queues and schedule the TP task of the matching access class to handle the transactions. The access class of the transaction returned by the MLS queue manager dictates which TP task will run next. When a task manager attempts to retrieve a transaction from an empty MLS queue it blocks on an event count and precipitates a process switch.

By delegating the management of a few similar tasks (identical save that each task only processes transaction of a given sensitivity level) within the process to the task manager, we hope to create an efficient mechanism for switching between the tasks, keeping kernel intervention to a minimum. Kernel intervention is only required when switching between processes (or transactions of different types.)

The final tier of the system contains the actual TP tasks which process the transactions. The tasks in this level are untrusted and are the objects that perform the work.

A. LAYERING AND DEPENDENCIES

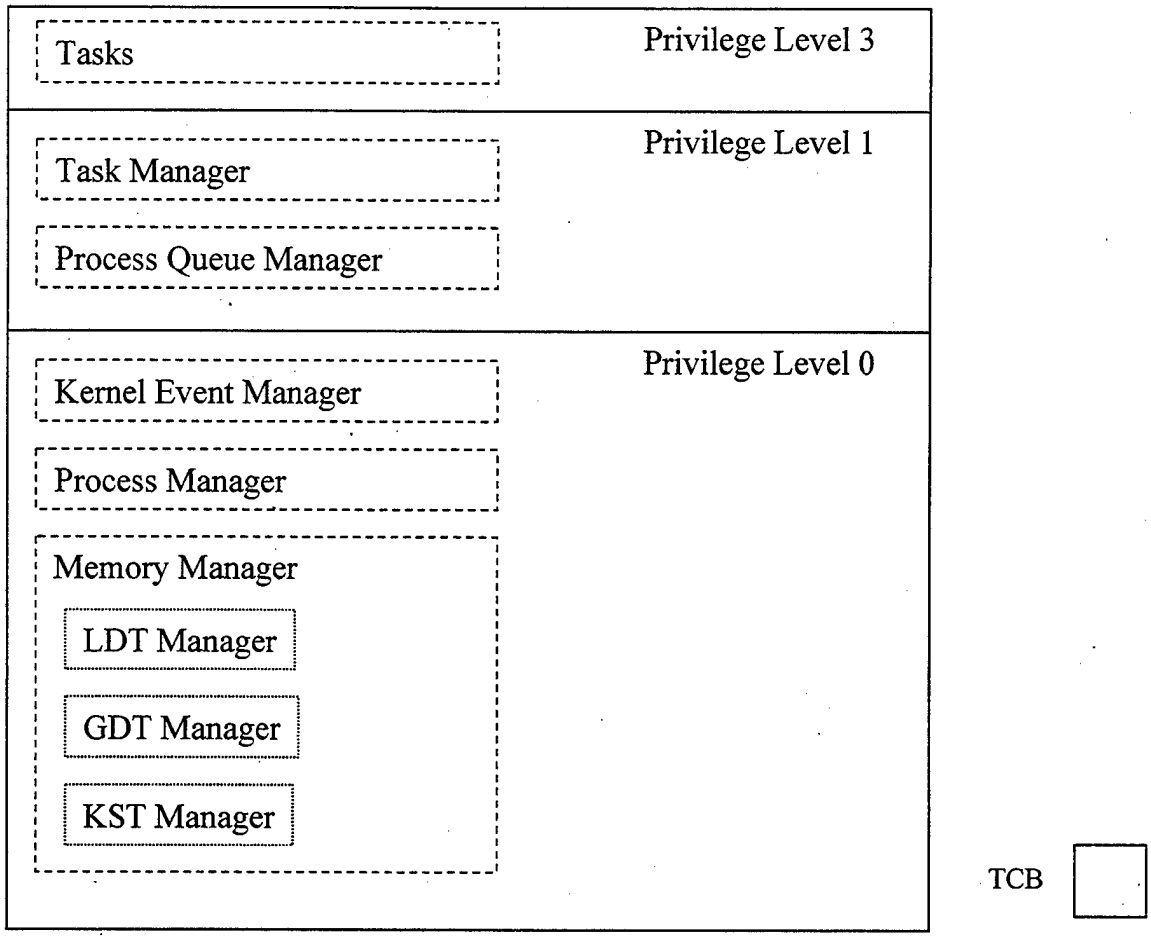

Figure 5.2: System Layering

Figure 5.2 shows the system layering and the trust boundary. The kernel resides in privilege level 0 (most privileged). The queue manager and task manager reside in privilege level 1. The untrusted TP tasks reside in privilege level 3. Privilege level 2 is not used in the current architecture and is available for future use. The kernel, queue manager and task manager are trusted. They have the ability to violate the chosen security policy but are trusted not to due to strict control over implementation. The TP tasks are not trusted and may be coded by anyone using whatever controls they choose. The TP tasks cannot violate the chosen security policy even if they try. This is not to say

that a poorly written or malicious TP task cannot cause the system to crash, it can. However, there is no way that any unauthorized information flows can occur to, from or between the TP tasks.

The layering pictured in Figure 5.2 is strict. Functions within a module only call functions in within themselves or within modules in a lower layer. In this avoid circular dependencies within the code.

B. SECURITY KERNEL

The security kernel consists of several distinct subsystems. There is a process manager, which is responsible for creating, scheduling and destroying processes. There is a memory manager, which manages the various local descriptor table (LDT) images (used by the individual tasks within each process space), the global descriptor table (GDT), memory allocation and deallocation, and adding or removing segments from any of the descriptor tables. Finally, there is a kernel event manager, which manages eventcounts used by the processes and the MLS queues. These eventcounts are used for synchronization by all modules in the system.

1. Process Manager

The process manager, as the name implies, manages the processes, which equate to different transaction types. The process manager implements several operations to manage processes.

- Create a process
- Switch from the active process to another process
- Destroy a process
- Change a process status
- Return an identifier for the current process

Associated with each process is a global descriptor table (GDT) image. These GDT entries are where each process stores the code and data the task manager needs to

perform its functions. Upon a process switch, this GDT image is switched to reflect the new processes' entries. The process manager also performs scheduling of processes. The process manager is supported by one data structure:

- Process table – holds the process information (including GDT image) needed to manage processes

The process table allows the kernel to map a given process to its GDT image. A process switch also results in a switch of the contents of the process portion of the GDT. Processes can be in one of three states: ready, running or blocked on an event.

2. Memory Manager

All code and data used by a task is contained wholly in the LDT and all code and data used by the kernel, the MLS queues and the processes are contained wholly in the GDT. As such, a task switch involves only an LDT switch. The memory manager can be viewed as being logically divided into two distinct subsystems; one which manages the LDT and one which manages the GDT. Whenever a new task is created, a new LDT image is created to hold the descriptors for its address space. Associated with each LDT image is an identifier which

a) LDT Manager

The LDT component maintains a database of LDT images. Each of these LDT images is associated with a given TP task. However, the mapping between a given LDT images and a specific task is not maintained by the kernel, but instead is maintained by the Task Managers. The LDT component provides functions to:

- Create a new LDT image
- Destroy an LDT image
- Add a segment to an LDT image
- Remove a segment from an LDT image
- Make a given LDT image the current LDT

When a task requests the addition of an item to its LDT, the request is brokered by the Task Manager. The Task Manager would add access class information and the specific LDT image identifier to the request prior to forwarding it to the kernel. The LDT component would check the security attributes of the segment being added to ensure they are consistent with the arguments of the request. If the request operation would not violate the security policy it is carried out. If the request operation would violate the security policy it is rejected.

When a task manager creates a new task, a new LDT image is created which the Task Manager would then associate with that task

The LDT component is supported by one data structure:

- LDT Database – keeps track of LDT images and their physical location in memory

b) GDT Manager

The GDT component of the Memory Manager is responsible for managing the GDT as well as allocating and deallocating memory. The specific functions it provides include:

- Create a new GDT image
- Destroy a GDT image
- Add a segment to a GDT image
- Remove a segment from a GDT image
- Switch GDT segments
- Allocate memory
- Deallocate memory

Associated with each process is a GDT image. On a process switch the process portion of the GDT must be saved to the currently running process' GDT image storage segment while the GDT image of the new process must be restored to the process portion of the GDT.

The call to switch a GDT segment comes from the Process Manager which actually performs the context switch.

The GDT manager is supported by one data structure:

- GDT Database – keeps track of the GDT images for the various processes

c) KST Manager

The KST manager is responsible for those functions that manipulate the Known Segment Table (KST). These functions include:

- Allocating a segment and adding it to the KST
- Deallocating a segment and removing it from the KST
- Returning the attributes associated with a segment

The KST manager is responsible for one data structure:

- KST – the Known Segment Table, keeps track of all segments currently in the system and their attributes (security label, descriptors, etc.)

3. Kernel Event Manager

The process event manager provides eventcounts for use by the processes and the MLS queues. The event manager provides the following functions:

- Create an eventcount
- Destroy an eventcount
- Wait on an eventcount
- Advance an eventcount
- Get a ticket

The MLS queues use kernel eventcounts to keep track of the number of items in a queue. A call to get work from an empty queue becomes a wait call on a kernel eventcount and leads to a process change. When new items are added to this queue, the associated kernel eventcount is advanced which moves the blocked process from the blocked process list to the ready process list making it eligible to be scheduled. Process scheduling is determined by the transaction flow through the MLS queues.

The kernel event manager is supported by one data structure:

- KED – the kernel event database, tracks the values of the various eventcounts and sequencers

C. PROCESS QUEUE MANAGER

The Process Queue Manager is the entity of the system which managers the MLS queues. It is layered between the process (Task Manager) and the kernel. The Process Queue Manager provides functions which allow processes to:

- Create an MLS queue
- Destroy an MLS queue
- Enqueue an item on an MLS queue
- Dequeue an item from an MLS queue
- Get and item from an MLS queue (whithout dequeuing it)

Initial input transactions are added to the system and are put on the MLS queues (usually by type) by some trusted input process. This process would enqueue an incoming transaction on the MLS queue associated with the process that handles transactions of that type. The transactions are of varying access classes. The Task Manager maintains single level tasks which process the transactions from the MLS queue (based on access class).

The Task Manager would make a 'get work' request to the MLS queue manager while providing a preferred access class. The Task Manager seeks to keep running the same TP task (at a specified access class) as long as work exists for that task to process. This minimizes task switches and provides maximum throughput of transaction through the system. The Process Queue Manager will return an item at the requested access class or an item at a different access class if: (1) there were no items at the requested access class or (2) there is a transaction with a higher priority than the next transaction of the requested access class. The access class of the item returned by the Process Queue

Manager determines whether the current task remains running or whether a new task will have to be scheduled.

A call to 'get work' from an empty queue would block (being translated into a wait call on a kernel eventcount). The process of the calling Task Manager would thus be blocked and a new process scheduled by the kernel. When the blocked process has an item enqueued to it (which also involves a call to advance the appropriate eventcount), the process would be moved to the ready list and could be scheduled to run.
All the functions of process queue manager are exported to the task manager.

The process queue manager is supported by one data structure:

- PQD – the process queue database, keeps track of information about the various MLS queues

D. TASK MANAGER

Each process contains a Task Manager which manages the single level TP tasks for each transaction type. The task manager creates, schedules and destroys the individual tasks.

The Task Manager is responsible for managing the single level tasks within each process space. The operations supported by the task manager include:

- Create a new single level task
- Switch from the current single level task to another
- Destroy a single level task

The Task Manager implements scheduling of single level tasks within its process space. Besides directly managing the TP tasks, the Task Manager also serves as an intermediary for task access to kernel memory management functions (add/remove from LDT).
When a tasks attempts to add or remove a segment from its LDT image, the Task Manager passes the request on to the kernel after adding the access class of the requesting

task and the identifier for its LDT image (both of which are kept track of by the Task Manager).

Additionally, the Task Manager interfaces with the Process Queue Manager to retrieve and insert items into MLS queues on behalf of the tasks. When a task makes a request for a new transaction, the Task Manager makes a call to 'get work' of that appropriate access class from the appropriate queue. If the returned item is of the requested access class, the Task Manager returns it to the TP task which continues to run (no task switch). If the Process Queue Manager should return an item of a differing access class, the Task Manager would suspend the current task and begin running the task of the access classs associated with the returned transaction. An attempt to 'get work' from an empty queue would block and not return until their were items available in the queue.

The Task Manager is supported by one data structure:

- TD – the task database, tracks the current single level tasks being managed by the task manager

E. TASKS

The tasks are in the outermost layer of the architecture and are the untrusted applications. It is the tasks that actually do whatever work is required by the transactions. Each task is at a single level and might coexist with copies of itself at different sensitivity levels within the same process space. However, through task manager manipulation of the LDT images, each task has it's own distinct address space (perhaps sharing a read-only code segment.) Since the task manager sets up an LDT image for a task when it is created, a task switch requires simply changing the current LDT.

Our chosen benchmark of performance will be transaction throughput. Our scheduling policy will entail completely processing all entries in a given queue before moving on to the next queue. So, a process will continue to execute so long as transactions remain in the MLS queue it is waiting on. Likewise, within the process, a

task continues to execute so long as transactions of the appropriate access class remain in the MLS queue its controlling process is waiting. In this manner, we minimize the number of task switches within a process and minimize the number of processes switches within the kernel. Each task will have a separate stack and a task switch will also involve a stack switch. By minimizing the frequency of process switches (which require the most time to accomplish) with our scheduling policy, and by making the task switches as quick as possible, we believe we can achieve better performance than would be possible implementing a TP system on top of a pre-existing MLS timesharing system.

F. INPUT/OUTPUT

We have not specifically addressed any input/output functionality required by the tasks besides the primitive enqueue, get_work and dequeue. In a real world application, the tasks might enqueue transactions to an MLS database. These transactions instruct the database to perform updates. Therefore, updating a database entails writing to the queue that services it.

Traditional block and character input/output and device support would also be useful additions to the system. Devices might be handled using the queues already in the system. Writing to a device would be abstracted to writing to the MLS queue associated with the device. Likewise, reading from a device would be equivalent to getting work from the MLS queue associated with the device.

G. DISTRIBUTED SCHEDULING

The system will support distributed scheduling and management. The tasks are managed almost wholly by the task manager which is distinct and separate from the kernel. When a task attempts to get_work from a queue which contains no items of the requested access class, it is blocked (using an eventcount) and another task within the same process space is scheduled (via a quick LDT switch). When all transactions within a given MLS queue are removed, the Task Manager's attempt to get_work would result

in a wait call on a kernel eventcount. The kernel would then schedule another process that did have work.

VI. KERNEL SPECIFICATION

The following sections contain the specification for the secure kernel, which controls the system memory, the management of the various processes and the kernel eventcounts. If consists of three components:

- Memory Manager (MM)
- Process Manager (PM)
- Kernel Event Manager (KEM)

A. MEMORY MANAGER (MM)

The memory manager is responsible for managing the GDT and LDT.

1. LDT Manager

The LDT component encapsulates those functions associated with management of the several LDT images, one associated with every task. It can be viewed as being divided into two parts, one that manages the database of LDT images and one that manages the LDT images themselves. The part that manages the LDT database provides functions to:

- Initialize the LDT database
- Create a new LDT image
- Switch to a specified LDT image
- Destroy a specified LDT image

The part of the LDT manager that manages the individual LDT images provides functions to:

- Initialize an LDT image
- Add a segment to a specified LDT image
- Removes a segment from a specified LDT image

a) LDT Manager Constants

a. LDT_DATABASE_SIZE – the size of the LDT database, this is maximum number of LDT images the system can manager at one time
b. LDT_DATABASE_MIN_ENTRY – the minimum value that can be used to index the LDT database
c. LDT_SIZE – this is the size of the LDT itself, in the Pentium architecture this is 8,120 descriptors (Intel, 1997).
d. LDT_SEGMENT_SIZE – the size (in bytes) of the LDT
e. END_OF_LIST – used to mark the end of the list of free database entries\

b) LDT Manager Databases

a. LDT Database

Each LDT entry stores information about one LDT image

```
struct LDT_Entry_Struct {
        selector : Selector;     // selector into the GDT for LDT image segment
        segment : Segment;       // segment id for LDT image
        free: Boolean;           // is this database entry in use?
        first_free: LDT_Entry;          // the first free item in the LDT
        next: LDT_Database_Entry;  // used to chain free entries together
};
```

The LDT database is an array of LDT entries. The indices into the LDT database array are the LDT identifiers.

```
struct LDT_Entry_Struct LDT_Database[LDT_DATABASE_SIZE];
```

c) LDT Manager Global Variables

a. Free_LDT_Entry : LDT_Database_Entry – The first free LDT entry in the LDT database, this entry points to the next free entry and so on. The last free entry would contain an END_OF_LIST.

d) LDT Manager System Calls

a. ldt_init_ldt_database

Procedure ldt_init_ldt_database()

Purpose: This functions initializes the entries of the LDT database

Inputs: None.

Outputs: None.

Processing:

```
{

// cycle through all LDT database entries
For index = LDT_DATABASE_MIN_ENTRY to LDT_DATABASE_SIZE
{
    // null all memory pointers
    LDT_Database[index].selector = NULL;
    LDT_Database[index].segment = NULL;
    // mark all entries as free
    LDT_Database[index].free = TRUE;
    // all entries are part of the free list
    LDT_Database[index].next = index + 1;
}
// the last entry is the end of the free list
```

```
LDT_Database[index].next = END_OF_LIST;
// the first free database slot is the first slot
Free_LDT_Entry = LDT_DATABASE_MIN_ENTRY;
// set return code and return
return_code = SUCCEEDED;
return return_code;
}
```

Effects:

- After completion, all entries in the LDT database are free, and all segment and descriptor pointers are null.

b. ldt_create_ldt

Fuction: ldt_create_ldt(ldt_access_class: Access_Class_Type, ldt_id: LDT_Database_Entry) : Sucess_Code_Type

Purpose: This call creates a new LDT image at a specified access class. The process is returned a index into the LDT database at the position where the entry for the new LDT image was placed.

Inputs:

- ldt_access_class : Access_Class_Type – The access class of the new LDT image. This value is provided by the task manager and corresponds to the access class of the associated task. This access class must be within the range allowed for the calling process.

Outputs:

- ldt_id: LDT_Entry – an index into the LDT database where the new LDT image was stored.
- Success_Code – indicates the status of the operation. Possible values include:
  - SUCCEEDED – The operation completed successfully
  - LDT_DATABASE_FULL – there is no free entries in the LDT database
  - NO_MEMORY – the function was unable to allocate the memory for the new LDT image segment

Processing

{

// Local variables

```
Success_Code        return_code;   // success or error code
Segment             ldt_segment;   // the segment which contains the LDT
Selector            ldt_selector;  // the selector for the LDT image in the GDT

return_code = SUCCEEDED;
// the default case is to return a null entry
ldt_id = NULL;
if (Free_LDT_List < LDT_DATABASE_MIN_ENTRY) {
        return_code = LDT_DATABASE_FULL); }
// allocate memory for the LDT, is unsuccessful, return the memory manager
// error code
else if (return_code = kst_allocate_memory(LDT_SEGMENT_SIZE,
        ldt_access_class,ldt_segment) != SUCCEEDED)
// add the allocated segment to the gdt
else if (return_code = gdt_add_to_gdt(ldt_segment,ldt_selector) !=
        SUCCEEDED) {
        // if we fail, deallocate the memory we just allocated
                kst_deallocate_memory(ldt_segment); }
else {
        ldt_id = Free_LDT_Entry;
        Free_LDT_Entry = LDT_Database[Free_LDT_Entry].next;
        LDT_Database[ldt_id].selector = ldt_selector;
        LDT_Database[ldt_id].segment = ldt_segment;
        LDT_Database[ldt_id].first_free = 0;
        LDT_Database[ldt_id].free = FALSE;
        return_code = ldt_init_ldt(ldt_id);
}
return return_code;
```

}

Effects:

- If successful, the KST will contain a new segment for the LDT image, the GDT will contain an entry for the LDT image segment and the LDT database will contain a new entry for the LDT image.

c. ldt_destroy_ldt

Function: ldt_destroy_ldt(ldt_id:LDT_Database_Entry) : Success_Code_Type

Purpose: This function destroys an LDT image. The entry in the LDT database is added to the free list and the memory used by the LDT image is deallocated.

Inputs:

- ldt_id: LDT_Database_Entry – an index into the LDT database, this is the entry that is destroyed.

Outputs:

- Success_Code – indicates the status of the operation. Possible values include:
- SUCCEEDED – The operation completed successfully
  - INVALID_LDT_DATABASE_ENTRY – The ldt id provided did not point to a valid database entry
  - DEALLOCATION_ERROR – an error occurred trying to deallocate the memory used by the LDT image

Processing:

```
{
        // Local variables
        Success_Code_Type  return_code;   // success or error code

        return_code = SUCCEEDED;
        // check for a valid database entry
        if (ldt_id < LDT_DATABASE_MIN_ENTRY OR
```

```
        ldt_id >= LDT_DATABASE_SIZE) {
        return_code = INVALID_LDT_DATABASE_ENTRY; }
    // check that the entry being used is not free
    else if (LDT_Database[ldt_id].free == TRUE) {
        return_code = INVALID_LDT_DATABASE_ENTRY; }
    // deallocate memory and return error code is unsuccessful
    else if (kst_deallocate_memory(LDT_Database[ldt_id].segment) !=
        SUCCEEDED) {
        return_code = DEALLOCATION_ERROR; }
    // remove from the GDT, return error code if unseccessful
    else if (gdt_remove_from_gdt(LDT_Database[ldt_id].selector) !=
        SUCCEEDED) {
        return_code = DEALLOCATION_ERROR; }
    // otherwise, remove the entry
    else {
        // add the current entry to the free list
        LDT_Database[ldt_id].next = Free_LDT_Entry;
        // mark as free
        LDT_Database[ldt_id].free = TRUE;
        // current entry becomes head of free list
        Free_LDT_Entry = ldt_id; }
    return return_code;
}
```

Effects:

- If successful, the entry in the KST for the LDT image segment will be deallocated, the memory used by the LDT image will be deallocated and the

LDT database entry occupied by the LDT image will be added to the free LDT database entry list and marked as free.

d. ldt_switch_ldt

Function: ldt_switch_ldt(ldt_id:LDT_Database_Entry) : Success_Code

Purpose: This function makes a specified LDT image the current LDT.

Inputs:

- ldt_id: LDT_Database_Entry – the LDT image to make the current LDT

Outputs:

- Success_Code – indicates the result of the operation. Possible values include:
  - SUCCEEDED – The operation completed successfully
  - INVALID_LDT_DATABASE_ENTRY – The ldt id provided does not point to a valid database entry

Processing:

```
{
    // Local variables
    Success_Code        return_code;   // success or error code

    return_code = SUCCEEDED;
    // check for valid LDT id
    if (ldt_id < LDT_DATABASE_MIN_ENTRY OR
       ldt_id >= LDT_DATABASE_SIZE) {
            return_code = INVALID_LDT_DATABASE_ENTRY; }
    // make sure the entry we are acting on is not marked free
    else if (LDT_Database[ldt_id].free == TRUE) {
```

```
            return_code = INVALID_LDT_DATABASE_ENTRY; }
        // switch the LDT register
        else {
            LDTR = LDT_Database[ldt_id].selector; }
        return return_code;
}
```

Effects:

- If successful, the LDTR will contain a selector for the LDT image specified.

e. ldt_add_to_ldt

Function: ldt_add_to_ldt(ldt_id:LDT_Database_Entry,segment:Segment, access:Access_Class,selector:Selector) : Success_Code

Purpose: This function adds a segment descriptor to the LDT image specified. A selector (index into the LDT) is returned.

Inputs:

- ldt_id: LDT_Database_Entry – identifies which LDT image the segment should be added to.
- segment: Segment – the segment ID which should be added to the LDT
- access: Access_Class – the access class of the task making the request

Outputs:

- selector: Selector – the selector (index into the LDT) of the segment just added
- Success_Code – indicates the result of the operation. Possible values include:
  - SUCCEEDED – the operation completed successfully
  - SECURITY_VIOLATION – added the specified segment to the specified LDT image would violate the security policy
  - UNKNOWN_SEGMENT – the segment specified does not correspond to a valid segment
  - INVALID_LDT_DATABASE_ENTRY – the LDT id provided does not index a valid database entry
  - LDT_FULL – the LDT is full, no additions are possible

Processing:

```
{
    // Local variables
    Success_Code        return_code;        // success or error code
    Access_Check        access_chk;         // access class of the LDT
    Selector            ldt_image_selector; // the selector for the LDT

    return_code = SUCCEEDED;
    // check for a valid LDT identifier
    if (ldt_id < LDT_DATABASE_MIN_ENTRY OR
                ldt_id >= LDT_DATABASE_SIZE) {
        return_code = INVALID_LDT_DATABASE_ENTRY; }
    else if (LDT_Database[ldt_id].free == TRUE) {
        return_code = INVALID_LDT_DATABASE_ENTRY; }
    access_chk = kst_access_check(segment, access);
    // check for a valid segment
    if (access_chk == UNKNOWN_SEGMENT) {
        return_code = UNKNOWN_SEGMENT; }
    // check for a policy violation
    else if (access_chk == SECURITY_VIOLATION) {
        return_code = SECURITY_VIOLATION; }
    // check for a full LDT
    else if (LDT_Database[ldt_id].first_free < 0) {
        return_code = LDT_FULL; }
    // otherwise, add the segment to the LDT
    else {
        selector = LDT_Database[ldt_id].first_free;
        ldt_image_selector = LDT_Database[ldt_id].selector;
```

```
            Get first free entry in LDT and set to first_free;
            Insert selector into LDT at location first_free;
        }
        return return_code;
}
```

Effects:

- If successful, the LDT image specified will have a new entry (descriptor) corresponding to the segment specified. The LDT image free list is updated to indicate that the entry just allocated is no longer free.

f. ldt_remove_from_ldt

Function: ldt_remove_from_ldt(ldt_id:LDT_Database_Entry,selector:Selector) : Success_Code

Purpose: This function removes a descriptor for a specified LDT image. This function does not deallocate the segment. The segment still exists and must be explicitly deallocated by the calling process with a call to mm_deallocate_memory.

Inputs:

- ldt_id: LDT_Database_Entry – an index in the LDT database specifying which LDT image the descriptor should be removed from.
- selector: Selector – the selector specifying which LDT entry is to be removed.

Outputs:

- Success_Code – Indicates the result of the operation. Possible values include:
- SUCCEEDED – The operation completed successfully.
  - INVALID_LDT_DATABASE_ENTRY – The LDT id provided does not point to a valid database entry.
  - INVALID_SELECTOR – The selector provided was invalid

Processing:

```
{
    // Local variables
    Success_Code        return_code;        // success or error code
```

```
    Selector                    ldt_image_selector;// LDT selector

    return_code = SUCCEEDED;
    // check for a valid LDT identifier
        if (ldt_id <  LDT_DATABASE_MIN_ENTRY OR
            ldt_id >= LDT_DATABASE_SIZE) {
            return_code = INVALID_LDT_DATABASE_ENTRY; }
        else if (LDT_Database[ldt_id].free == TRUE) {
        return_code = INVALID_LDT_DATABASE_ENTRY; }
    // check for a valid selector
    else if (selector < MIN_LDT_ENTRY OR selector >= LDT_SIZE) {
        return_code = INVALID_SELECTOR; }
    // otherwise, remove the item from the LDT
    else {
        ldt_image_selector = LDT_Database[ldt_id].selector;
        *(ldt_image_selector:selector) = LDT_Database[ldt_id].first_free;
        LDT_Database[ldt_id].first_free = selector;
        return_code = SUCCEEDED; }
    return return_code;
}
```

Effects:

- If successful, the specified selector in the specified LDT image is added to the free entry list.

2. GDT Manager

The GDT manager of the memory manager handles those functions that manipulate the GDT and GDT images. This component provides the following services:

- Create a new GDT image
- Switch between GDT images
- Destroy a specified GDT image
- Add a segment to the GDT
- Removes a segment from the GDT
- Allocate a block of memory
- Deallocate a block of memory
- Returns the descriptor associated with a segment id

The GDT is logically divided into two sections. One part of the GDT holds descriptors used by the kernel and MLS queues (this part remains static during process switches) and the other part holds descriptors used by the processes (this part gets swapped on a process switch).

a) GDT Manager Constants

a. GDT_IMAGE_SIZE – Size of the process section of the GDT.
b. GDT_DATABASE_SIZE – The number of entries in the database that tracks process GDT images
c. KERNEL_GDT_SIZE – Size of the kernel section of the GDT.
d. PROCES_GDT_SIZE – Size of the process section of the GDT
e. GDT_DATABASE_MIN – The minimum value used to index the GDT database.
f. END_OF_LIST – Used to denote the end the list of free entries
g. GDT_MIN – The minimum value used to index the GDT itself (on Pentium architectures this will usually be 0).
h. KERNEL_GDT – Indicates that kernel section of the GDT is being acted on.

i. PROCESS_GDT – Indicates that the process section of the GDT is being acted on.

b) GDT Manager Databases

a. Process GDT Database (GDT_DATABASE)

This database stores information about the GDT image associated with each process. Unlike the LDT, where a switch only requires the changes of one register value, a GDT switch requires the copying of the current values of the GDT into a GDT image and the copying of a new GDT image into the GDT.

Each GDT_DATABASE entry stores information about one process GDT image

```
struct GDT_Database_Entry {
    selector: Selector;                  // the GDT selector for the GDT image
    segment: Segment;                    // segment id for the GDT image
    free_list[GDT_IMAGE_SIZE]: Integer;  // free entry map
    first_free: Integer;                 // index to first free entry
    next: GDT_Database_Entry;            // link to next entry in free list
}
```

The process GDT database stores information about each process GDT image. The GDT image id corresponds to an index into the array.

```
struct GDT_Database_Entry GDT_DATABASE[GDT_DATABASE_SIZE];
```

b. Kernel GDT Free Table

This database keeps a map of the free and available entries in the kernel section of the GDT. Is simply a linked list stored as an array.

This database maps the free slots in the kernel section of the GDT

KERNEL_GDT_FREE[KERNEL_GDT_SIZE] Integer;

c) GDT Manager Global Variables

a. GDT_DATABASE_FIRST_FREE – Points to the first GDT database entry that is available. This entry then points to the next free entry and so forth. An entry of NIL indicates that there are no free entries.

b. KERNEL_GDT_FIRST_FREE – Points to the first free GDT slot in the kernel section of the GDT. This is an index into the KERNEL_GDT_FREE table.

d) GDT Manager System Calls

a. gdt_create_gdt

Function: gdt_create_gdt(gdt_id:GDT_Database_Entry): Success_Code

Purpose: This function creates a new process GDT image storage area and passes back the if associated with the GDT image.

Inputs: None

Outputs:

- gdt_id: GDT_Database_Entry – the index into the GDT database associated with the process GDT image just created.
- Success_Code – indicates the results of the operation. Possible values include:
  - SUCCEEDED – the operation competed successfully
  - GDT_DATABASE_FULL – there are no free slots in the GDT database
  - NO_MEMORY – the function was unable to allocate memory for the new GDT image

Processing:

```
{
    // Local variables
    Sucdess_Code    return_code;   // stores return code
    Segment         gdt_segment;   // stores GDT image segment id
    Selector        gdt_selector;  // stores GDT image GDT selector
```

```
return_code = SUCCEEDED;
gdt_id = NULL;// initialize return parameter to NULL
// check for free entry in the GDT database
if (GDT_DATABASE_FIRST_FREE < GDT_DATABASE_MIN) {
        // if none, set return code
        return_code = GDT_DATABASE_FULL; }
// else, try to allocate memory for the GDT segment
else if { (kst_allocate_memory(GDT_IMAGE_SIZE,SYSTEM_CLASS,
            gdt_segment)        != SUCCEEDED)
        // if the memory allocation fails, set return code
                return_code = NO_MEMORY; }
// else, try to add the just allocated segment to the kernel GDT
        else if { (gdt_add_to_gdt(ldt_segment,KERNEL_GDT,
            gdt_selector) != SUCCEEDED)
        // if the attempt to add to the kernel's GDT fails, set return code
            return_code = NO_MEMORY; }
// otherwise, initialize the GDT image and return
else {
        // the current free GDT entry is the one we use, and we set the free
        // entry to the next in the list
        gdt_id = GDT_DATABASE_FIRST_FREE;
        GDT_DATABASE_FIRST_FREE =
                        GDT_Database[GDT_DATABASE_FIRST_FREE].next;
        // initialize the current GDT entry
        GDT_Database[gdt_id].selector = gdt_selector;
        GDT_Database[gdt_id].segment = gdt_segment;
        // initialize the free list for the new GDT image
```

```
        for (index = 1 to GDT_SIZE) {
                GDT_Database[gdt_id].free_list[index] = index+1; }
                GDT_Database[gdt_id].free_list[index] = END_OF_LIST;
                GDT_Database[gdt_id].first_free = GDT_MIN;
                // This is an active entry, so the next points to nothing
                GDT_Database[gdt_id].next = END_OF_LIST;
        }
    return return_code;
}
```

Effects:

- If successful, the GDT database will contain a new entry corresponding to a new process GDT image. All entries in the new GDT will be marked as free and the index associated with the new GDT image will be returned to the caller.

b. gdt_destroy_gdt

Function: gdt_destroy_gdt(gdt_id:GDT_Database_Entry): Success_Code

Purpose: This function destroys a GDT image. The entry in the GDT database is added to the free list and the memory used by the GDT image is deallocated.

Inputs:

- gdt_id: GDT_Database_Entry – an index into the GDT database indicating which GDT image should be deallocated.

Outputs:

- Success_Code – indicates the result of the operation. Possible values include:
  - SUCCEEDED – The operation completed successfully
  - INVALID_GDT_ENTRY – The gdt it provided is not valid
  - DEALLOCATION_ERROR – An error occurred trying to deallocate the memory used by the GDT image

Processing:

```
{
        // Local variables
        Success_Code return_code;   // stores the return code

        return_code = SUCCEEDED;
        // Check for a gdt_id within the proper range
        if (gdt_id < GDT_DATABASE_MIN) OR
                (gdt_id >= GDT_DATABASE_SIZE) {
```

```
            // if invalid, set the return code
            return_code = INVALID_GDT_ENTRY; }
        // otherwise, try to deallocate the memory used by the GDT image
        else if (kst_deallocate_memory(GDT_Database[gdt_id].segment) !=
            SUCCEEDED) {
            // if unsucceessful, set the return code
            return_code = DEALLOCATION_ERROR; }
        // attempt to remove the descriptor from the GDT
        else if (kst_remove_from_gdt(KERNEL_GET,
            GDT_Database[gdt_id].selector) != SUCCEEDED) {
            // if unsuccessful, set the return code
            return_code = DEALLOCATION_ERROR; }
        // finally, if we get this far, update the GDT database to remove the entry
        else {
            // add the GDT entry to the beginning of the free list
            GDT_Database[gdt_id].next = GDT_DATABASE_FIRST_FREE;
            GDT_DATABASE_FIRST_FREE = gdt_id;
            return_code = SUCCEEDED;
        }
        return return_code;
}
```

Effects:

- If successful, the memory used by the GDT image is deallocated, the segment is removed from the Known Segment Table, the selector for the GDT image is removed from the GDT and the entry in the GDT database is added to the list of available entries.

c. gdt_switch_gdt

Function: gdt_switch_gdt(old_gdt_id:GDT_Database_Entry, new_gdt_id:GDT_Database_Entry): Success_Code

Purpose: This function swaps out the current process GDT image to the specified GDT image storage area and swaps in the process GDT image from the specified GDT image storage area.

Inputs:

- old_gdt_id: GDT_Database_Entry – the GDT image witch should be swapped out. Note, the memory manager has no way of knowing which process is currently running, thus the caller of this routine must specify the currently running process GDT image area.
- new_gdt_id: GDT_Database_Entry – the GDT image to swap in.

Outputs:

- Success_Code – indicates the result of the operation. Possible values include:
  - SUCCEEDED – The operation completed successfully
  - INVALID_GDT_DATABASE_ENTRY – One of the GDT database entries provided does not point to a valid entry in the GDT database.

Processing:

```
{
    // Local variables
    Success_Code return_code;   // stores the return code
```

```
    return_code = SUCCEEDED;
    // Check for an GDT database index input out of range
    if (old_gdt_id < GDT_DATABASE_MIN) OR
            (old_gdt_id > GDT_DATABASE_SIZE) OR
            (new_gdt_id < GDT_DATABASE_MIN) OR
            (new_gdt_id > GDT_DATABASE_SIZE)
        // Set return code to indicate invalid entry
        return_code = INVALID_GDT_DATABASE_ENTRY; }
    // Otherwise, perform the swap
    else {
        // Copy the contents of the GDT process image to the old process area
        memcopy(GDT_IMAGE_SELECTOR:0,
            GDT_DATABASE[old_gdt_id].selector:0, GDT_IMAGE_SIZE);
        // Copy the contents of the new GDT process image into the GDT
        memcopy(GDT_DATABASE[new_gdt_id].selector:0,
            GDT_IMAGE_SELECTOR:0, GDT_IMAGE_SIZE);
    }
    // return success code
    return return_code;
}
```

Effects:

- If successful, the current contents of the GDT process area will be copied to the specified GDT process image storage area and the contents of the new GDT process image storage area will be copied into the GDT process area.

d. gdt_add_to_gdt

Function: gdt_add_to_gdt(segment:Segment, gdt_id:GDT_Database_Entry, gdt_selector: Selector) : Success_Code

Purpose: This function adds a specified segment to either the process GDT or the kernel GDT (the two parts of the physical GDT). A segment can only be added to the current process GDT image since this function acts on the GDT and not any of the images stored elsewhere. The kernel area of the GDT requires not special handling since it remains static during process switches.

Inputs:

- segment: Segment – the segment id (index into the KST) for the segment that should be added the GDT
- gdt_id: Segment – the identifier for the GDT image to add the segment to. If this value is set to KERNEL_GDT, that indicates that the segment should be added to the kernel section of the GDT otherwise the segment is added to the process section of the GDT.

Outputs:

- selector: Selector – a selector into the GDT for the segment just added
- Success_Code – Indicates the results of the operation. Possible values include:
  - SUCCEEDED – The operation completed successfully.
  - NO_MEMORY – The specified GDT is full
  - INVALID_GDT_DATABASE_ENTRY – The GDT id is invalid
  - INVALID_SEGMENT – The segment provided does not correspond to a real segment

Processing:

```
{
        // Local variables
        Success_Code        return_code;   // stores the return code
        Descriptor          seg_desc;      // the descriptor to add

        // Check for gdt_id within range
        if (((gdt_id < GDT_DATABASE_MIN) OR
                (gdt_id >= GDT_DATABASE_SIZE))
                AND   (gdt_id != KERNEL_GDT)) {
                // if invalid, set the return code
                return_code = INVALID_GDT_DATABASE_ENTRY; }
        // otherwise, check for an available entry if kernel GDT
        else if ((gdt_id == KERNEL_GDT) AND
         (KERNEL_GDT_FIRST_FREE < GDT_DATABASE_MIN)) {
         // if no free entries in kernel GDT
         return_code == NO_MEMORY; }
        // otherwise, check for an available entry if we are adding to the
        // process GDT
        else if ((gdt_id != KERNEL_GDT) AND
         (GDT_Database[gdt_id].first_free < GDT_DATABASE_MIN)) {
         // if no free entries in process GDT
         return_code == NO_MEMORY; }
        // check for a valid segment by trying to get its associated descriptor
        else if ((seg_desc = mm_get_descriptor(segment)) !=
         SUCCEEDED)) {
```

```
        // if we can't get the descriptor, then the seg id is invalid
        return_code = INVALID_SEGMENT; }
    // finally, is we can actually update the GDT
    else {
        // if we are adding to the kernel section of the GDT
        if (gdt_id == KERNEL_GDT) {
            Assign the first free slot in the kernel GDT to selector;
            Update the kernel GDT free list;
            Add the descriptor to GDT at selector;
        // otherwise, we are adding to the process GDT
        else {
            Assign the first free slot in the kernel GDT to selector;
            Update the kernel GDT free list;
            Add the descriptor to GDT at selector;
        }
    }
    return return_code;
}
```

Effects:

- If successful, the descriptor for the segment passed in is added to either the kernel or process section of the GDT and the appropriate free list is updated. A selector for the GDT entry added is passed back to the caller.

e. gdt_remove_from_gdt

Function: gdt_remove_from_gdt(get_id: GDT_Database_Entry,selector:Selector): Success_Code

Purpose: This function removes an entry from the GDT, either the kernel or process section. The slot occupied by the removed selector is added to the appropriate free list. This function only removes the selector, the segment and the memory it occupies must be explicitly deallocated by the caller.

Inputs:

- get_id: GDT_Database_Entry – The id of the GDT image to delete the selector from, or the kernel GDT
- selector: Selector – the selector to remove from the GDT

Outputs:

- Success_Code – indicates the result of the operation. Possible values include:
  - SUCCEEDED – the operation completed successfully.
  - INVALID_GDT_DATABASE_ENTRY – the GDT image specified is invalid
  - INVALID_SELECTOR – the selector specified is invalid

Processing:

```
{
   // Local variables
   Success_Code return_code;   // stores the return code

   return_code = SUCCEEDED;
```

```
// First, check for a valid gdt_id
return_code = SUCCEEDED;
if ((gdt_id != KERNEL_GDT) AND ((gdt_id) < GDT_DATABASE_MIN)
    OR (gdt_id >= GDT_DATABASE_SIZE)) {
    // if invalid entry, set return code
    return_code = INVALID_GDT_DATABASE_ENTRY; }
// otherwise, if we are operating on the kernel GDT, check for
// valid selector range
if ((gdt_id == KERNEL_GDT) AND
    (selector < GDT_MIN) OR
    selector >= KERNEL_GDT_SIZE)) {
    // if selector out of range for kernel table, set return code
    return_code = INAVLID_SELECTOR; }
// otherwise, if we are operating on the process GDT, check for
// valid selector range
if ((gdt_id != KERNEL_GDT) AND
        (selector < GDT_MIN  OR
        selector >=  PROCESS_GDT_SIZE)) {
        // if selector is out of range, set return code
        return_code = INVALID_SELECTOR; }
// otherwise, actually remove the selector
else {
        // if we are operating on the kernel GDT
        if (gdt_id == KERNEL_GDT) {
                Add the selected GDT entry to the kernel GDT free list;
                Initialize the descriptor so it is no longer valid;
        // otherwise, remove from process GDT
                Add the selected GDT entry to the process GDT free list;
```

```
                Initialize the descriptor so it is no longer valid;
            }
        }
        return return_code;
}
```

Effects:

- If successful, this process will blank a selector from the GDT (either kernel or process sections) and add the blanked slot to the list of available GDT slots.

3. KST Manager

The KST manager of the memory manager handles those functions that manipulate the KST. This component provides the following services:

- Allocate a block of memory
- Deallocate a block of memory
- Returns the descriptor associated with a segment id

a) KST Manager Constants

a. KST_MIN – Minimum value used to index the KST.
b. KST_MAX – Maximum value used to index the KST
c. KST_SIZE – The size of the KST.

b) KST Manager Databases

a. Known Segment Table (KST)

This table maps segment id's to their associated descriptors and security attributes.

```
// The KST is an array of KST entry records. The index into the KST serves
// as the segment identified.

struct KST_Entry KST[KST_SIZE];

// Each KST entry stores information about one segment

struct KST_Entry {
        descriptor: Descriptor;          // descriptor associated with this segment
        access: Access_Class; // access class associated with segment
```

```
    next: KST_Entry;            // next KST entry in free list
}
```

c) KST Manager Global Variables

a. KST_FIRST_FREE – Points to the first free KST slot. If this index is less than 0, we can no longer add segment to the system.

d) KST Manager System Calls

a. kst_allocate_memory

Function: kst_allocate_memory(size:Integer, access_class:Access_Class, segment:Segment) : Success_Code

Purpose: This functions allocates a block of memory of the appropriate size and creates a descriptor that is added to the known segment table. The index into the know segment table that corresponds to the added segment is returned to the caller.

Inputs:

- size: Integer – the size of the memory block to allocated (in bytes).
- access_class: Access_Class – the label to associated with the newly created segment

Outputs:

- semgnet: Segment – the segment id associated with the newly created segment
- Success_Code – indicates the result of the operation. Possible values include:
  - SUCCEEDED – the operation completed successfully
  - NO_MEMORY – there is no memory to allocate
  - KST_FULL – there are no more available entries in the KST

Processing:

```
{
        // Local variables
        Success_Code return_code=NULL;  // stores return code
```

```
        Descriptor      seg_desc;               // descriptor for the new segment

        return_code = SUCCEEDED;
        // make sure we have an available slot in the KST
        if (KST_FIRST_FREE < KST_MIN) {
                // if not, set return code
                return_code = KST_FULL; }
        // otherwise, create and assign the descirptor
        else {

                // ***** allocate physical memory at this point, create
                // ***** descriptor and assign to seg_desc.  If there
                // ***** isn't enough physical memory, set return to
                // ***** NO_MEMORY

                // if memory was successfully allocated
                if (return_code != NO_MEMORY) {
              // assign the segment number
                segment = KST_FIRST_FREE;
                        KST_FIRST_FREE = KST[KST_FIRST_FREE].next;
                // assign descriptor and access class to new segment
                        KST[segment].descriptor = seg_desc;
                        KST[segment].access = access_class;
                }
        }
        return return_code;
}
```

Effects:

- If successful, physical memory is allocated and a descriptor is created, this descriptor is added to the know segment table with the supplied access class. The index into the KST which identifies the new segment is passed back to the caller.

b. kst_deallocate_memory

Function: kst_deallocate_memory(segment:Segment) : Success_Code

Purpose: This function deallocates memory associated with a segment id and adds the KST slot to the list of free slots

Inputs:

- segment: Segment – the index into the KST which identifies the segment to be deallocated

Outputs:

- Success_Code – indicates the result of the operation. Possible values include:
  - SUCCEEDED – the operation completed successfully
  - INVALID_SEGMENT – the segment provided is invalid
  - DEALLOCATION_ERROR – an error occurred during deallocation of memory

Processing:

```
{
    // Local variables
    Success_Code return_code = NULL; // save return code

    // check for valid segment
    if (segment < KST_MIN) OR (segment > KST_MAX) {
            // if invalid, set return code
            return_code = INVALID_SEGMENT; }
```

```
    // otherwise, perform the deallocation
    else {

        // ***** Deallocate the physical memory associated with
        // ***** the descriptor KST[segment].descriptor.  If there
        // ***** is an error performing the deallocation, set
        // ***** return_code to DEALLOCATION_ERROR.  Also,
        // ***** deallocated memory should be wiped to prevent
        // ***** reuse

        // if memory was successfully deallocated
        if (return_code != DEALLOCATION_ERROR) {
            // add segment to free list
            KST[segment].next = KST_FIRST_FREE;
            KST_FIRST_FREE = segment;
            // blank descriptor and access to prevent reuse
            KST[segment].descriptor = BLANK_DESCRIPTOR;
            KST[segment].access = NO_ACCESS;
        }
    }
    return return_code;
}
```

Effects:

- If successful, the KST entry for the specified segment should be blanked and that entry added to the list of available entries. Also, the memory associated with the segment should be deallocated.

c. kst_get_descriptor

Function: kst_get_descriptor(segment:Segment, seg_desc:Descriptor) : Success_Code

Purpose: This functions returns the descriptor associated with a segment in the KST.

Inputs:

- segment: Segment – the segment id for which the descriptor is sough

Outputs:

- seg_desc: Descriptor – the descriptor associated with the requested segment
- Success_Code – indicates the result of the operation. Possible values include:
  - SUCCEEDED – the operation completed successfully
  - INVALID_SEGMENT – the request segment is invalid

Processing:

```
{
    // Local variables
    Success_Code return_code;    // stores the return code

    // check for a valid segment id
    if (segment < 0) OR (segment >= MAX_KST_SIZE) {
            // if invalid, set return code
            return_code = INVALID_SEGMENT; }
    // otherwise, return the descriptor
    else {
            seg_desc = KST[segment].descriptor;
```

```
            // set return code
            return_code = SUCCEEDED;
        }
        return return_code;
}
```

Effects:

- This system call does not alter the state of the system if successful.

B. PROCESS MANAGER (PM)

The process manager is responsible for managing the multiple processes in the system. Some of its functions include:

- Create new processes
- Destroys processes
- Switches processes
- Changes process status
- Return the current process

1. Process Manager Constants

a. MAX_PROCESSES – The maximum number of processes the system can manage.
b. MIN_PROCESS – The least value that can be used to index the process table.
c. MAX_PROCESS – The maximum value that can be used to index the process table.

2. Process Manager Databases

a. Process Database

The process database keeps information about the process currently in the system. This includes context, current status, and GDT image. The process id is the index into this table.

The process database is an array of process entries. It is set to the maximum number of processes the system can handle

```
struct Process_Entry Process_Database[MAX_PROCESSES];
```

Process entries store information about each process

```
struct Process_Entry {
    gdt_id: GDT_Database_Entry;      // the GDT image for this process
    status: Process_Status;                  // the current process status
    context: Process_Context;        // the saved process context
    previous: Process_Entry;         // previous process entry in list
    next: Process_Entry;             // next process entry in list
}
```

3. Process Manager Global Variables

a. FREE_PROCESS_HEAD & FREE_PROCESS_TAIL – point to the head and tail of the list of free process entries

b. READY_PROCESS_HEAD & READY_PROCESS_TAIL – point to the head and tail of the list of ready processes

c. BLOCKED_PROCESS_HEAD & BLOCED_PROCESS_TAIL – points to the head and tail of the list of blocked processes

d. CURRENT_PROCESS – the index of the entry for the running process

4. Process Manager System Calls

a. pm_switch_process

Function: pm_switch_process()

Purpose: This function runs the next ready process and suspends the current process.

Inputs: None.

Outputs: None.

Processing:

```
{
    // Local variables
    Process_Entry current_process;      // the currently running process
    Process_Entry next_process;         // the next process to be run

    return_code = SUCCEEDED;
    // Save the current processes context – the caller should handle putting
    // the current process on the appropriate blocked queue
    save_context(Process_Database[CURRENT_PROCESS].context);
    // Wait until there is a ready process available
    while((next_process = READY_PROCESS_HEAD) > MIN_PROCESS);
    // Switch the process GDT images
    mm_switch_gdt(Process_Database[CURRENT_PROCESS].gdt_id,
```

```
            Process_Database[next_process].gdt_id;
    // Move new process from READY to RUNNING
    pm_change_process_status(next_process,RUNNING);
    // Restore the new process context
    restore_context(Process_Database[next_process].context);
    // **** magic point – at this point we are in the new process;
    return;
}
```

Effects:

- When this process returns, the system will be running the next available ready process.

b. pm_create_process

Function: pm_create_process(process parameters[TBD],new_process:Process_Entry): SuccessCode

Inputs: TBD.

Outputs:

- new_process: Process_Entry – the entry in the process database corresponding to the new process added
- SuccessCode – Indicates the result of the operation. Possible values include:
  - SUCCEEDED – the operation completed successfully
  - PROCESS_TABLE_FULL – there are no more available entries in the process table
  - GENERAL_ERROR – the process could not be created due to some other condition

Processing:

```
{
  // Local variables
  Process_Entry        new_process;  // the id of the new process
  Success_Code         return_code;  // holds the return code
  GDT_Database_Entry gdt_id;         // holds GDT image for new process

  return_code = SUCCEEDED;
  // Check for available slots in the process database
  if (FREE_PROCESS_HEAD < 0) {
```

```
            // set return code to indicate no free slots
            return_code = PROCESS_TABLE_FULL }
    // Otherwise, Initialize a new GDT image for this process
    else if (gdt_id = mm_create_gdt(gdt_id) != SUCCEEDED) {
            // if we can't create a gdt image, set error code
            return_code = GENERAL_ERROR; }
    // **** Perform other checks as needed
    else if ( ...) { }
    // otherwise, initialize the entry
    else {
            // assign the process id
            new_process = FREE_PROCESS_HEAD;
            // assign the process GDT image
            Process_Database[new_process] = gdt_id;
            // **** Initialize memory as needed
            // Add the new process to the ready list
            if (pm_change_process_status(new_process,READY) !=
                    SUCCEEDED) {
                    // if we couldn't make the process ready, error
                    return_code = GENERAL_ERROR }
    }
    return return_code;
}
```

Effects:

- If successful, the function will remove an process entry from the free list, initialize a GDT image and other memory, and add the new process to the ready list.

c. pm_destroy_process

Function: pm_destroy_process(process_id:Process_Entry): SuccessCode

Inputs:

- process_id: Process_Entry – the process to destroy

Outputs:

- Success_Code – indicates the result of the operation. Possible values include:
  - SUCCEEDED – the operation succeeded
  - INVALID_PROCESS – the process id provided was not valid
  - DEALLOCATION_ERROR – an error occurred deallocating memory
  - FAILED – the operation failed

Processing:

```
{
    // Local variables
    Success_Code return_code;    // holds the return code

    return_code = SUCCEEDED;
    // Make sure the process id is valid
    if ((process_id < MIN_PROCESS) OR (process_id > MAX_PROCESS)
            OR (Process_Database[process_id] = FREE)) {
            // the operation fails if the id is invalid
            return_code = INVALID_PROCESS; }
    // Try to deallocate the memory used by the GDT image
```

```
else if (mm_destory_gdt(Process_Database[process_id].gdt_id !=
        SUCCEEDED) {
// if we can't, we have an error
return_code = DEALLOCATION_ERROR; }
        // otherwise, remove the process
        else {
                // move the entry to the free list
                if (pm_change_process_status(process_id,FREE) !=
                        SUCCEEDED) {
                        // if we can't, we have an error
                        return_code = FAILED; }
        }
        return return_code;
}
```

Effects:

- If successful, this function will remove the specified process from whichever list it is currently on, deallocate its GDT image and will move the process entry to the free list.

d. pm_change_process_status

Function: pm_change_process_status(process_id: Process_Entry, new_status:Process_Status): Success_Code

Purpose: This function changes the status of a specified process.

Inputs:

- process_id: Process_Entry – the process we want to change the status of.
- new_status: Process_Status – the new status for the process

Outpust:

- Success_Code – indicated the result of the operation. Possible values include:
- SUCCEEDED – the operation succeeded
  - INVALID_STATUS – the new status is not valid
  - INVALID_PROCESS – the process in not valid
  - FAILED – the operation failed for some other reason

Processing:

```
{
    // Local variables
    Success_Code        return_code; // holds the return code
    Process_Entry       prev;   // temp used for list manipulation
    Process_Entry       next;   // temp used for list manipulation

    return_code = SUCCEEDED;
```

```
// Checks for a process id within bounds
if ((process_id < MIN_PROCESS) OR (process_id > MAX_PROCESS)) {
        // if out-of-bound, set error code
        return_code = INVALID_PROCESS; }
// check for a valid status
else if (new_process NOT IN {RUNNING,FREE,BLOCKED,READY}) {
        // if not a valid status, set error code
        return_code = INVALID_STATUS; }
// otherwise, make the change
else {
        // first, if the new status is the same as the old status we are done
        if (NOT(Process_Database[process_id].status = new_status)) {
                // otherwise, remove the process from the old list
                switch Process_Database[process_id].status {
                        // if current process is running, it doesn't have to
                        // be removed from any list
                        case RUNNING:
                                break;
                        // item is on the blocked list
                        case BLOCKED:
                                // if the previous item in the list is empty,
                                // we are at the head
                                if (Process_Database[process_id].previous
                                        < MIN_PROCESS) {
                                        BLOCKED_PROCESS_HEAD =
                        Process_Database[process_id].next; }
                                // if the next item in the list is empty, we are at the
                                // tail
```

```
            if (Process_Database[process_id].next <
                    MIN_PROCESS) {
                    BLOCKED_PROCESS_TAIL =
Process_Database[process_id].previous; }
            // if not at the head or the tail, remove item from list
            if (NOT(Process_Database[pcoress_id].next
                    < MIN_PROCESS) AND
            NOT(Process_Database[process_id].previous <
                    MIN_PROCESS)) {
                    next = Process_Database[process_id].next;
                    prev =
                    Process_Database[process_id].previous;
                    Process_Database[prev].next = next;
                    Process_Database[next].previous = prev; }
            break;
// process is on the ready list
case READY:
            // if previous item in the list is empty, we are at the
            // head
            if (Process_Database[process_id].previous <
                    MIN_PROCESS) {
                    READY_PROCESS_HEAD =
                    Process_Database[process_id].next; }
            // if the next item in the list is empty, we are at the
            // tail
            if (Process_Database[process_id].next <
                    MIN_PROCESS) {
                    READY_PROCESS_TAIL =
```

```
                Pprocess_Database[process_id].previous; }
        // if at the head or the tail, remove item from list
        if (NOT(Process_Database[pcoress_id].next <
                MIN_PROCESS)
                AND
                NOT(Process_Database[process_id].previou
        s < MIN_PROCESS)) {
                next = Process_Database[process_id].next;
                prev =
                 Process_Database[process_id].previous;
                Process_Database[prev].next = next;
                Process_Database[next].previous = prev; }
        break;
// process is on the free list
case FREE:
        // if previous item in the list is empty, we are at the
        // head
        if (Process_Database[process_id].previous <
                MIN_PROCESS) {
                FREE_PROCESS_HEAD =
Process_Database[process_id].next; }
        // if the next item in the list is empty, we are at the
        // tail
        if (Process_Database[process_id].next <
                MIN_PROCESS) {
                FREE_PROCESS_TAIL =
                Pprocess_Database[process_id].previous; }
        // if at the head or the tail, remove item from list
```

```
                    if (NOT(Process_Database[pcoress_id].next <
                            MIN_PROCESS) AND
                    NOT(Process_Database[process_id].previous <
                            MIN_PROCESS))
                            next = Process_Database[process_id].next;
                            prev =
                             Process_Database[process_id].previous;
                            Process_Database[prev].next = next;
                            Process_Database[next].previous = prev; }
                    break;
            // otherwise, we have an error
            case default:
                    return_code = FAILED;
}
// now we add the process to the new list
switch new_status {
        // if we are making the process running, don't do anything
        case RUNNING:
                break:
        // working on the blocked list
        case BLOCKED:
                // if new list is empty, set head to new process
                if (BLOCKED_PROCESS_HEAD =
  BLOCKED_PROCESS_TAIL = -1) {
  BLOCKED_PROCESS_HEAD = process_id;
  BLOCKED_PROCESS_TAIL = process_id; }
                // otherwise, just add to the end of the list
                else {
```

```
Process_Database[BLOCKED_PROCESS_TAIL].next
                    = process_id;
                                Process_Database[process_id].next =
                                        END_OF_LIST;
                                Process_Database[process_id].previous =
                    BLOCKED_PROCESS_TAIL;
                                BLOCKED_PROCESS_TAIL = process_id;
                        }
                        break;
                    // working on the ready list
                    case READY:
                        // if new list is empty, set head to new process
                        if (READY_PROCESS_HEAD =
                READY_PROCESS_TAIL = END_OF_LIST) {
                READY_PROCESS_HEAD = process_id;
                READY_PROCESS_TAIL = process_id; }
                        // otherwise, just add to the end of the list
                        else {

Process_Database[READY_PROCESS_TAIL].next =
                                        process_id;
                                Process_Database[process_id].next =
                                        END_OF_LIST;
                                Process_Database[process_id].previous =
                    READY_PROCESS_TAIL;
                                READY_PROCESS_TAIL = process_id;
                        }
```

```
                    break;
                // working on the FREE list
                case FREE:
                    // if new list is empty, set head to new process
                    if (FREE_PROCESS_HEAD =
            FREE_PROCESS_TAIL = -1) {
            FREE_PROCESS_HEAD = process_id;
            FREE_PROCESS_TAIL = process_id; }
                    // otherwise, just add to the end of the list
                    else {

    Process_Database[FREE_PROCESS_TAIL].next =
                                process_id;
                            Process_Database[process_id].next =
                                END_OF_LIST;
                            Process_Database[process_id].previous =
                        FREE_PROCESS_TAIL;
                                FREE_PROCESS_TAIL = process_id;
                    }
                    break;
                case default:
                    return_code = FAILED;
            }
            // change the process status field
            Process_Database[process_id].status = new_status;
        }
    }
return return_code;
```

}

Effects:

- If successful, this function will move a given process from one process status list to another and rest the process status field

C. KERNEL EVENT MANAGER (KEM)

The process event manager controls the kernel eventcounts. These eventcounts are used for synchronization and scheduling (through the queue managers.) The functions provided include:

- Create a new eventcount
- Delete an eventcount
- Advance an eventcount
- Wait on an eventcount
- Create a new sequencer
- Delete a sequencer
- Get a ticket from a sequencer

1. Kernel Event Manager Constants

a. MAX_EVCT – The number of kernel eventcounts the system will support.

b. MIN_EVCT – The smallest eventcount identifier

c. MAX_SEQ – The number of kernel sequencers the system will support.

d. MIN_SEQ – The smallest sequencer identifier

e. END_OF_LIST – The end of the free list.

f. INIT_VALUE – The value used to initialize eventcounts and sequencers

2. Kernel Event Manager Databases

a. Kernel Event Database

The kernel event database keeps track of the kernel eventcounts and their values.

// Kernel event entries store information about each eventcount

```
struct Kernel_Event_Entry {
        Integer:        count;  // the value of the eventcount
        // a list of processes which are blocked on this eventcount
        List of <process_id, wait_value> pairs: blocked;
        Evct_Status :   statue; // The eventcount statue
        Integer:        next;   // chains the free list
}
```

// The kernel eventcounts

Array of Kernel_Event_Entry : KED[MIN_EVCT..MAX_EVCT];

b. Kernel Sequencer Database

The kernel sequencer database keeps track of the sequencers and their values

Kernel sequencer entries store information about each sequencer

```
struct Kernel_Seq_Entry {
    Integer:        value;  // the value of the sequencer (unsigned)
    Evct_Status:    status; // the status of the eventcount
    Integer:        next;   // chains the free list together
}
```

The sequencer database stores the sequencers

Array of Kernel_Seq_Entry : KSD[MIN_SEQ..MAX_SEQ];

3. Kernel Event Manager Variables

a. KED_Free_Head – The first free slot in the KED

b. KSD_Free_Head – The first free slot in the KSD

4. Kernel Event Manager Functions

a. ked_create_evct

Function: ked_create_evct(Integer: evct): Success_Code

Purpose: Allocates a new evenctcount, initializes it and returns it to the caller.

Inputs: None.

Outputs:

- Integer: evct – An index into the KED which is the identifier for the new eventcount
- Success_Code – Indicates the result of the operation. Possible values include:
  - SUCCEEDED – The operation completed successfully
  - NONE_AVAILABLE – No empty eventcounts are available
  - FAILED – The operation was unable to complete successfully

Processing:

```
{
        // Local variables
        Success_Code        return_code;   // success or error code

        return_code = SUCCEEDED;
        if (KED_Free_Head == END_OF_LIST) {
                // set error code
```

```
            return_code = NONE_AVAILABILITY; }
        // otherwise, allocate the eventcount
        else {
            evct = KED_Free_Head;
            KED[evct].value = 0;
            KED_Free_Head = KED[evct].next;
        }
        return return_code;
}
```

Effects:

- If successful, a new eventcount is allocated, initialized and returned to the caller.

b. ked_destroy_evct

Function: ked_destroy_evct(Integer: evct): Success_Code

Purpose: This function destroys a specified eventcount by returning it to the free list.

Inputs:

- Integer : evct – The identifier for the evenctounc we would like to delete

Outputs:

- Success_Code – Indicates the result of the operation. Possible values include:
  - SUCCEEDED – The operation completed successfully
  - INVALID_EVCT – The specified eventcount is not valid

- FAILED – The operation did not complete successfully

Processing:

```
{
        // Local variables
        Success_Code        return_code;   // success or error code

        return_code = SUCCEEDED;
        // check for valid eventcount identifier
        if (evct < MIN_EVCT OR evct > MAX_EVCT) {
                return_code = INVALID_EVCT; }
        else if (KED[evct].status ≠ IN_USE) {
                return_code = FAILED; }
        // otherwise, deallocate the eventcount
        else {
                KED[evct].next = KED_Free_Head;
                KED_Free_Head = evct;
                return_code = SUCCEEDED;
        }
        return return_code;
}
```

Effects:

- If successful, the specified eventcount is added to the free list and is no longer available.

c. ked_advance_evct

Function: ked_advance_evct(Integer: evct): Success_Code

Purpose: This function advances the specified eventcount and will unblock any processes which are waiting on the new eventcount value

Inputs:

- Integer: evct – The eventcount to advance.

Outputs:

- Success_Code – Indicated the result of the operation. Possible values include:
  - SUCCEEDED – The operation completed successfully
  - INVALID_EVCT – The specified eventcount was not valid
  - FAILED – The operation did not complete successfully.

Processing:

```
{
        // Local variables
        Success_Code  return_code; // success or error code

        return_code = SUCCEEDED;
        // check for valid eventcount identifier
        if (evct < MIN_EVCT OR evct > MAX_EVCT) {
                return_code = INVALID_EVCT; }
        else if (KED[evct].status ≠ IN_USE) {
                return_code = INVALID_EVCT; }
```

```
        else {
                // advance the eventcount
                KED[evct].value = KED[evct].value + 1;
                for each <process_id, wait_value> pair in KED[evct].blocked {
                        // if a process is waiting on the new value, unblock it
                        if (wait_value <= KED[evct].value) {
                                pm_change_process_status(process_id, READY);
                                remove <process_id, wait_value> from
                                        KED[evct].blocked;
                        }
                }
        }
        return return_code;
}
```

Effects:

- If successful, the specified eventcount is advanced and any processes which were waiting on it are unblocked.

d. ked_wait_evct

Function: ked_wait_evct(Integer: evct, Integer: value): Success_Code

Purpose: This function waits on a specified value for an evenctcount.

Inputs:

- Integer : evct – The eventcount we would like to wait on.
- Integer : value – The value of the eventcount we would like to wait on

Outputs:

- Success_Code – Indicated the result of the operation. Possible value inbluce:
  - SUCCEEDED – The operation completed successfully.
  - INVALID_EVCT – The specified eventcount identifier was not valid
  - FAILED – The operation did not complete successfully

Processing:

```
{
        // Local variables
        Success_Code        return_code;   // success or error code
        Integer             process_id;    // the current process

        return_code = SICCEEDED;
        // check for valid eventcount identifier
        if (evct < MIN_EVCT OR evct > MAX_EVCT) {
                return_code = INVALID_EVCT; }
```

```
    else if (KED[evct].status ≠ IN_USE) {
            return_code = INVALID_EVCT; }
    else {
            // if the eventcount has not yet reached the requested value,
            // block the current process
            if (value > KED[evct].value) {
                    process_id = pm_get_current_process();
                    pm_change_process_status(process, BLOCKED);
                    add <process_id, value> pair to KED[evct].blocked;
                    pm_switch_process();
            }
    }
    return return_code;
}
```

Effects:

- If successful, the function returns when the specified eventcount has reached the specified value.

e. ksd_create_seq

Function: ksd_create_seq(Integer: seq): Success_Code

Purpose: Allocates a new sequencer, initializes it and returns it to the caller.

Inputs: None.

Outputs:

- Integer: seq – An index into the KSD which is the identifier for the new sequencer
- Success_Code – Indicates the result of the operation. Possible values include:
  - SUCCEEDED – The operation completed successfully
  - NONE_AVAILABLE – No empty sequencer are available
  - FAILED – The operation was unable to complete successfully

Processing:

```
{
        // Local variables
        Success_Code        return_code;   // success or error code

        return_code = SUCCEEDED;
        if (KSD_Free_Head == END_OF_LIST) {
                // set error code
                return_code = NONE_AVAILABLE; }
        // otherwise, allocate the sequencer
        else {
                seq = KSD_Free_Head;
```

```
                KSD[seq].value = INIT_VALUE;
                KSD_Free_Head = KSD[seq].next;
                return_code = SUCCEEDED;
            }
        }
```

Effects:

- If successful, a new sequencer is allocated, initialized and returned to the caller.

f. ksd_destroy_seq

Function: ksd_destroy_seq(Integer: seq): Success_Code

Purpose: This function destroys a specified sequencer by returning it to the free list.

Inputs:

- Integer : seq – The identifier for the evenctounc we would like to delete

Outputs:

- Success_Code – Indicates the result of the operation. Possible values include:
  - SUCCEEDED – The operation completed successfully
  - INVALID_SEQ – The specified sequencer is not valid
  - FAILED – The operation did not complete successfully

Processing:

```
{
    // Local variables
    Success_Code        return_code;   // success or error code

    return_code = SUCCEEDED;
    // check for valid sequencer identifier
    if (seq < MIN_SEQ OR seq > MAX_SEQ) {
            return_code = INVALID_SEQ; }
    else if (KSD[seq].status ≠ IN_USE) {
            return_code = FAILED; }
    // otherwise, deallocate the sequencer
```

```
    else {
            KSD[seq].next = KSD_Free_Head;
            KSD_Free_Head = seq;
            return_code = SUCCEEDED;
    }
return return_code;
}
```

Effects:

- If successful, the specified sequencer is added to the free list and is no longer available.

g. ksd_get_ticket

Function: ksd_advance_seq(Integer: seq, Integer: ticket): Success_Code

Purpose: This function advances a sequencer and returns the new value.

Inputs:

- Integer: seq – The sequencer to get a ticket from

Outputs:

- Success_Code – Indicated the result of the operation. Possible values include:
  - SUCCEEDED – The operation completed successfully
  - INVALID_SEQ – The specified sequencer was not valid
  - FAILED – The operation did not complete successfully.

Processing:

```
{
        // Local variables
        Success_Code        return_code; // success or error code

        return_code = SUCCEEDED;
        // check for valid sequencer identifier
        if (seq < MIN_SEQ OR seq > MAX_SEQ) {
                return_code = INVALID_SEQ; }
        else if (KSD[seq].status ≠ IN_USE) {
                return_code = INVALID_SEQ; }
        else {
```

```
            // advance the sequencer
            KSD[seq].value = KSD[seq].value + 1;
            // set the return value
            ticket = KSD[seq].value;
            return_code = SUCCEEDED;
        }
        return return_code;
}
```

Effects:

- If successful, the specified sequencer is advance and the new value is returned.

VII. PROCESS QUEUE MANAGER SPECIFICATION

The process queue manager is responsible for maintaining the queues that service the various processes and TP tasks. The functions it provides include:

- Create a queue
- Delete a queue
- Enqueue an item to a queue
- Dequeue an item from a queue
- Get work from a queue (without dequeing the item)

Each queue is an MLS queue. The elements of the queue are the IDs of the segments that contain the transactions. The queues are organized based upon priority. Additionally, each queue also contains per access class pointers that permit per-access class searches of the queue.

A request to get work from a queue contains a requested access class. The QTM will return a segment id and an access class. If a queue element exists with a higher priority than the next element at the requested access class, the high priority element will be returned along with its access class. If no elements exist at the requested access class, then the highest priority element on the queue along with its access class will be returned.

The PQM essentially implements the chosen task scheduling policy. In this case, the chosen policy is to continue processing transactions of a given access class from a specific MLS queue until a higher priority transaction enters the queue or we exhaust all queue transactions of a given access class. When no items remain in the queue, it blocks on a wait call to a kernel eventcount which precipitates a process switch.

This specification implements all queues as arrays. This often creates inefficient insertion and search algorithms as well as leading to possible deadlock. A practical implementation would implement the priority queue using an efficient, dynamic datastructure (such as a heap).

A. PROCESS QUEUE MANAGER CONSTANTS

a. MAX_MLS_QUES – The maximum number of MLS queues the system can handle
b. MAX_QUE_SIZE – The maximum size of an MLS queue
c. MAX_ACCESS_CLASSES – The maximum number of access classes in the system
d. MIN_MQUE – The minimum value used to index the list of MLS queues
e. MIN_QUE – The minimum value used to index the queue itself

B. PROCESS QUEUE MANAGER DATABASES

a. Process Queue Database (PQD)

Queue elements have the following format:

```
struct Queue_Entry {
    Access_Class: access;       // access class of this element
    Integer:      priority;         // priority of this element
    Segment:      seg_id;       // segment id of the transaction
    Integer:      next_pri;     // next element in the priority queue
    Integer:      prev_pri;         // previous element in the priority queue
    Integer:      next_access;  // next item in the access class queue
    Integer:      prev_access;  // previous item in the access class queue
}
```

The process queue database stores information about each MLS process queue.

Each entry in the process queue database has the following format

```
struct Process_Queue_Entry {
    Kernel_Evct: eventcount;            // the eventcount associated with this queue
    Queue_Entry: mls_queue[1..MAX_QUE_SIZE]; // the queue elements
    // pointers to elements within the queue, creates our per access class queues
    // we store both heads and tails.  An access class indexes into this array
    Integer: que_heads[1..MAX_ACCESS_CLASSES];
    Integer: que_tails[0..MAX_ACCESS_CLASSES];
    // pointers to the queue head and tail
    Integer: pri_head;                  // head of the priority queue
    Integer: pri_tail;                  // tail of the queue
    Integer: free_head;                 // head of the free element within the queue
    Integer: hold_head;                 // head of queue
    Integer: hold_tail;
    // used to chain together free entries in the PQD
    Integer: next_free;
    Status_Code: que_status;      // holds the status of the queue
}
```

The process queue database

```
struct Process_Queue_Entry PQD[MAX_MLS_QUES];
```

C. PROCESS QUEUE MANAGER MODULE VARIABLES

a. PQD_FIRST_FREE – index into the PQD of the first free entry. This would in turn be chained to the other free entries.

D. PROCESS QUEUE MANAGER FUNCTIONS

a. pqm_create

Function: pqm_create(MLS_Queue_ID: mls_q): Success_Code

Purpose: This function sets up a new MLS queue and returns an index into the PQD which identifies the new queue.

Inputs: None.

Outputs:

- MLS_Queue_ID: mls_q – the index into the PQD which identifies the new queue
- Success_Code – indicates the result of the operation. Possible values include:
  - SUCCEEDED – the operation completed successfully
  - NO_RESOURCES – the operation failed due to an inability to allocate necessary resources

Processing:

{

```
// Local variable
Success_Code:     return_code;   // holds the return code
Kernel_EVCT:      k_evct;        // kernel eventcount for new queue
Integer:          ix;            // loop index variable

return_code = SUCCEEDED;
// check for a free entry in the PQD
if (PQD_FIRST_FREE == EMPTY_LIST) {
        // if none, set error code
        return_code = NO_RESOURCES; }
// otherwise, get a kernel eventcount for this queue
else if (kem_create(k_evct) != SUCCEEDED) {
        // if we couldn't create a kernel eventcount, set error
        return_code = NO_RESOURCES; }
// otherwise, set up the queue
else {
        mls_q = PQD_FIRST_FREE;
        // advance the free list
        PQD_FIRST_FREE = PQD[PQD_FIRST_FREE].next_free;
        // initialize queue attributes
        PQD[mls_q].eventcount = k_evct;
// initialize each element of the queue (create the free list)
// the field used to point to the next item in the priority list
// 'pri_next' is used for the free list link as well
        for (ix = 1; ix <= MAX_QUE_SIZE; ix = ix + 1) {
                PQD[mls_q].mls_queue[ix].next_pri = ix + 1; }
        // initialize the access class pointers (all empty)
        for (ix = 1; ix <= MAX_ACCESS_CLASSES; ix = ix + 1) {
```

```
                PQD[mls_q].que_heads[ix] = EMPTY_QUE;
                PQD[mls_q].que_tails[ix] = EMPTY_QUE;
            }
            // initialize other pointers
            PQD[mls_q].pri_head = EMPTY_QUE;
            PQD[mls_q].pri_tail = EMPTY_QUE;
            PQD[mls_q].free_head = EMPTY_QUE;
            PQD[mls_q].hold_head = EMPTY_QUE;
            PQD[mls_q].hold_tail = EMPTY_QUE;
            PQD[mls_q].que_status = IN_USE;
            // we're done
    }
    return return_code;
}
```

Effects:

- If successful, a new MLS queue will be setup with all its elements on marked free. The calling process will be returned the index into the PQD corresponding to the new queue.

b. pqm_delete

Function: pqm_delete(MLS_Queue_ID: mls_q): Success_Code

- Purpose: This function removes a queue from the PQD. This function will only work if the queue is empty. It is the responsibility of the calling program to remove all items from the queue prior to attempting to delete it.

Inputs:

- MLS_Queue_ID: mls_q – the PQD index identifying the queue to delete.

Outputs:

- Success_Code – indicated the result of the operation. Possible value include:
  - SUCCEEDED – the operation completed successfully
  - NOT_EMPTY – the queue in question is not empty.
  - INVALID_QUEUE – the queue referenced is not currently in use

Processing:

```
{
    // Local variables
    Success_Code:       return_code;        // holds the return code

    return_code = SUCCEEDED;
    // Check for a valid MLS queue id
    if ((mls_q < MIN_MQUE) OR (mls_q > MAX_MLS_QUES)) {
            // if invalid, set error code
            return_code = INVALID_QUEUE; }
```

```
    // check that the referenced queue is in use
    else if (PQD[mls_q].queue_status != IN_USE) {
            // if invalid, set error code
            return_code = INVALID_QUEUE; }
    // make sure the queue is empty
    else if (PQD[mls_q].pri_head != EMPTY_QUE) {
            // if not empty, set error code
            return_code = NOT_EMPTY; }
    // otherwise, go ahead an deallocate the queue index
    else {
            PQD[mls_q].queue_status = FREE;
            PQD[mls_q].next_free = PQD_FIRST_FREE;
            PQD_FIRST_FREE = mls_q;
            // we're done
    }
    return return_code;
}
```

Effects:

- If successful, the entry in the PQD indicated is returned to the list of free PQD entires.

c. pqm_enque

Function: pqm_enque(MLS_Queue_ID:mls_q, Segment: seg_id, Integer: priority): Success_Code

Purpose: This function adds an item to a MLS queue.

Inputs:

- MLS_Queue_ID: mls_q – the MLS queue to which we want to add an item
- Segment: seg_id – the identifier for the segment we would like to add to the queue, this segment contains the transaction
- Integer: priority – the priority of the item to be added

Outputs:

- Success_Code – indicates the success of the operation. Possible values include:
  - SUCCEEDED – the operation completed successfully
  - NO_RESOURCES – a resource needed to complete the operation was unavailable
  - INVALID_ARG – an argument provided was not valid

Processing

```
{
    // Local variable
    Success_Code return_code;          // the return code
    Access_Class: access;              // the access class of the segment
```

```
Integer:    que_item;          // the new item in the queue
Integer:    ix;                // loop index variable
Integer     prev_item;         // holds queue item index
Boolean:    done = FALSE;      // completion flag

return_code = SUCCEEDED;
// check for a valid MLS que
if ((mls_q < MIN_MQUE) OR (mls_q > MAX_MLS_QUES)) {
        // set error code
        return_code = INVALID_ARG; }
else if (PQD[mls_q].queue_status != IN_USE) {
        // set error code
        return_code = INVALID_ARG; }
// check for a valid segment
else if (mm_get_access(seg_id, access) != SUCCEEDED) {
        // set error code
        return_code = INVALID_ARG; }
// otherwise, add the item
else {
        // check for free slot in the queue
        if PQD[mls_q].free_head == EMPTY_LIST {
                // if none avail, set error code
                return_code = NO_RESOURCES; }
        }
        // otherwise, add the item
        else {
                que_item = PQD[mls_q].free_head;
                PQD[mls_q].free_head =
```

```
PQD[mls_q].queue[PQD[mls_q].free_head].next_pri;
PQD[mls_q].queue[que_item].priority = priority;
PQD[mls_q].queue[que_item].access = access;
PQD[mls_q].queue[que_item].seg_id = seg_id;
// First, insert into the priority queue
ix = PQD[mls_q].pri_tail;
while ((ix != EMPTY_END) AND (done != TRUE)) {
        // check if we are in the right priority spot
        if (PQD[mls_q].queue[ix].priority < priority) {
                // if not, check the next item
    ix = PQD[mls_q].queue[ix].prev_pri ; }
        // we are at the right spot to insert the item
        else {
                done = TRUE;
        }
}
// Now, we actually insert the item in the priority queue
// check to see if we reached the end of the queue
if (ix == EMPTY_QUEUE) {
        PQD[mls_q].queue[que_item].next_pri =
                PQD[mls_q].que_head;
        PQD[mls_q].queue[que_item].prev_pri = EMPTY_QUE;
        PQD[mls_q].pri_head = que_item;
        // check for an empty queue
        if (PQD[mls_q].pri_tail = EMPTY_QUEUE) {
                PQD[mls_q].pri_tail = que_item;
        }
}
```

```
// otherwise, we stopped somewhere in the middle of the queue
else {
        prev_item = PQD[mls_q].queue[ix].prev_pri;
        // see if the item we are pointing to has an item before it
        if (prev_item != EMPTY_QUE) {
                // if yes, set its forward link
                PQD[mls_q].queue[prev_item].pri_next =
                que_item;
        }
        // insert the new item into the priority queue
        PQD[mls_q].queue[ix].prev_pri = que_item;
        PQD[mls_q].queue[que_item].next_pri = ix;
        PQD[mls_q].queue[que_item].prev_pri = prev_item;
}
// Second, search the priority queue
ix = PQD[mls_q].que_tails[access];
while ((ix != EMPTY_END) AND (done != TRUE)) {
        // check if we are in the right priority spot
        if (PQD[mls_q].queue[ix].priority < priority) {
                // if not, check the next item
     ix = PQD[mls_q].queue[ix].prev_access ; }
        // we are at the right spot to insert the item
        else {
                done = TRUE;
        }
}
// Now, we actually insert the item in the access class
// queue
```

```
                // Check to see if we reached the end of the queue
                if (ix == EMPTY_QUEUE) {
                        PQD[mls_q].queue[que_item].next_access =
                                PQD[mls_q].que_head;
                PQD[mls_q].queue[que_item].prev_access = EMPTY_QUE;
                PQD[mls_q].pri_head = que_item;
                // check for an empty queue
                if (PQD[mls_q].pri_tail = EMPTY_QUEUE) {
                PQD[mls_q].pri_tail = que_item;
                }
            }
            else {
                prev_item = PQD[mls_q].queue[ix].prev_access;
                // see if the item we are pointing to has an item before it
                if (prev_item != EMPTY_QUE) {
                        // if yes, set its forward link
                        PQD[mls_q].queue[prev_item].pri_access = que_item;
                }
                // insert the new item into the priority queue
                PQD[mls_q].queue[ix].prev_pri = que_item;
                PQD[mls_q].queue[que_item].next_access = ix;
                PQD[mls_q].queue[que_item].prev_access = prev_item;
            }
        }
    }
    return return_code;
}
```

Effects:

- If successful, this function inserts a new item into the queue. Both priority links and access class links are updated. The item is inserted into the queues based upon priority.

d. pqm_deque

Function: pqm_deque(MLS_Queue_ID:mls_q, Segment: seg_id): Success_Code;

Purpose: This function removes an item from a specified MLS queue.

Inputs:

- MLS_Queue_ID: mls_q – the MLS queue from which to dequeue the item.
- Segment: seg_id – the segment to remove from the queue.

Outputs:

- Success_Code – indicates the result of the operation. Possible values include:
  - SUCCEEDED – the operation was successful.
  - INVALID_QUE – the MLS queue specified was not valid
  - ITEM_NOT_FOUND – the specified item was not found on the queue

Processing:

```
{
   // Local variables
   Success_Code      return_code;   // stores the return code
   Integer           ix;            // loop index variable
```

```
Access_Class    access;    // the item's access class
Boolean         done;      // loop termination flag
Integer         item;      // stores temporary index into queue

return_code = SUCCEEDED;
if ((mls_q < 0) OR (mls_q > MAX_MLS_QUES)) {
    // set error code
    return_code = INVALID_QUE; }
else if (PQD[mls_q].queue_status != IN_USE) {
    // set error code
    return_code = INVALID_QUE; }
// otherwise, search for the item
else {
        // check the hold queue first.
        ix = PQD[mls_q].hold_head;
        done = FALSE;
        while (ix != EMPTY_QUE) {
                if (PQD[mls_q].queue[ix].seg_id == seg_id) {
                        done=TRUE;
                }
                else {
                        ix = PQD[mls_q].queue[ix].next_pri;
                }
        }
        // check if we found the item
        if (ix != EMPTY_QUE) {
                // first, remove from priority queue
                item = PQD[mls_q].queue[ix].prev_pri;
```

```
        // check for beginning of queue
        if (item == EMPTY_QUE) {
                PQD[mls_q].hold_head = PQD[mls_q].queue[ix].next_pri;
        }
        else {
                PQD[mls_q].queue[item].next_pri =
            PQD[mls_q].queue[ix].next_pri;
        }
        // check for end of queue
        item = PQD[mls_q].queue[ix].next_pri;
        if (item == EMPTY_QUE) {
                PQD[mls_q].hold_tail = PQD[mls_q].queue[ix].prev_pri;
        }
        else {
                PQD[mls_q].queue[item].prev_pri =
            PQD[mls_q].queue[ix].prev_pri;
        }
}
// if we found the item, skip searching the priority que
if (done != TRUE) {
while (ix != EMPTY_QUE) {
        if (PQD[mls_q].queue[ix].seg_id == seg_id) {
                done=TRUE;
        }
        else {
                ix = PQD[mls_q].queue[ix].next_pri;
        }
}
```

```
// check if we found the item
        if (ix != EMPTY_QUE) {
                // first, remove from the priority queue
                item = PQD[mls_q].queue[ix].prev_pri;
                // check for beginning of queue
        if (item == EMPTY_QUE) {
                        PQD[mls_q].pri_head =
                                PQD[mls_q].queue[ix].next_pri;
                }
                else {
                PQD[mls_q].queue[item].next_pri =
                  PQD[mls_q].queue[ix].next_pri;
        }
                // check for end of queue
                item = PQD[mls_q].queue[ix].next_pri;
                if (item == EMPTY_QUE) {
                PQD[mls_q].pri_tail = PQD[mls_q].queue[ix].prev_pri;
        }
                else {
                        PQD[mls_q].queue[item].prev_pri =
                  PQD[mls_q].queue[ix].prev_pri;
                }
        }
}
// if we found the item, update the access class links
if (done == TRUE) {
        item = PQD[mls_q].queue[ix].prev_access;
        // check for beginning of queue
```

```
            if (item == EMPTY_QUE) {
                        PQD[mls_q].que_heads[access] =
                          PQD[mls_q].queue[ix].next_access;}
            else {
                        PQD[mls_q].queue[item].next_access =
                    PQD[mls_q].queue[ix].next_access;
                }
                // check for end of queue
                item = PQD[mls_q].queue[ix].next_access;
                if (item == EMPTY_QUE) {
                PQD[mls_q].que_tails[access] =
                    PQD[mls_q].queue[ix].prev_access;
            }
                else {
                      PQD[mls_q].queue[item].prev_access =
                    PQD[mls_q].queue[ix].prev_access;
                }
                // add removed item to free list
                PQD[mls_q].queue[ix].next_pri = PQD[mls_q].free_head;
                PQD[mls_q].free_head = ix;
            }
            else {
                return_code = ITEM_NOT_FOUND;
            }
        }
        return return_code;
}
```

Effects:

- If successful, the queue element which contains the specified segment ID is removed from the specified MLS queue.

e. pqm_get_work

Function: pqm_get_work(MLS_Queue_ID: mls_q, Access_Class: access, Segment: seg_id): Success_Code

Purpose:

This function returns the segment ID of an item from the specified MLS queue. The access class passed in might also be changed if the returned item is of a different access class (due to high priority or no items of the requested access class).

Inputs:

- MLS_Queue_ID: mls_q – the MLS queue from which we would like to get work
- Access_Class: access – the requested access class, if work exists at this access class without any higher priority elements, it is returned. This input is also an output as it might be changed by the function

Outputs:

- Access_Class: access – contains the access class of the returned item
- Segment: seg_id – contains the segment ID of the return item, if any
  - Success_Code – indicates the result of the operation. Possible values include:
  - SUCCEEDED – the operation completed successfully
  - INVALID_QUE – the specified MLS queue is invalid

Processing:

```
{
    // Local variables
    Success_Code return_code;                    // stores the return code
    Access_Class     return_access;              // the access class of return item
    Integer          item, pri_item, access_item; // temporary queue indices
    Integer          ticket;                     // holds the ticket for the queue

    return_code = SUCCEEDED;
    // check for a valid MLS que
    if ((mls_q < MIN_MQUE) OR (mls_q > MAX_MLS_QUES)) {
            // set error code
            return_code = INVALID_QUE; }
    else if (PQD[mls_q].queue_status != IN_USE) {
            // set error code
            return_code = INVALID_QUE; }
    // otherwise, get an item
    else {
            // block if there are no items on the queue
            kem_get_ticket(PQD[mls_q].evct,ticket);
            kem_wait(PQD[mls_q].evct,ticket);
            if (PQD[mls_q].que_heads[access] == EMPTY_QUE) {
                    item = PQD[mls_q].pri_head; }
            else {
                    pri_item = PQD[mls_q].pri_head;
                    access_item = PQD[mls_q].que_heads[access];
                    if (PQD[mls_q].queue[pri_item].priority >
                    PQD[mls_q].queue[access_item].priority) {
                    item = pri_item;
```

```
        else {
                item = access_item;
        }
}
access = PQD[mls_q].queue[item].access;
// remove item from priority and access class links
PQD[mls_q].pri_head = PQD[mls_q].queue[item].next_pri;
if (PQD[mls_q].pri_tail = item) {
        PQD[mls_q].pri_tail = EMPTY_QUE;
}
else {
        pri_prev = PQD[mls_q].queue[item].prev_pri;
        PQD[mls_q].queue[pri_prev].next_pri = EMPTY_QUE;
}
PQD[mls_q].que_heads[access] =
  PQD[mls_q].queue[item].next_access;
if (PQD[mls_q].que_tails[access] = item) {
        PQD[mls_q].que_tails[access] = EMPTY_QUE;
}
else {
        access_prev = PQD[mls_q].queue[item].prev_access;
        PQD[mls_q].queue[pri_prev].next_access = EMPTY_QUE;
}
// add item to hold queue,
pri_tail = PQD[mls_q].hold_tail;
PQD[mls_q].hold_tail = item;
PQD[mls_q].queue[item].prev_pri = EMPTY_QUE;
if (pri_tail != EMPTY_QUE) {
```

```
                PQD[mls_q].queue[item].next_pri = pri_tail;
                PQD[mls_q].queue[pri_tail].prev_pri = item;
            }
            seg_id = PQD[mls_q].queue[item].seg_id;
        }
        return return_code;
}
```

Effects:

- If successful, this call will return a segment ID and an access class. The returned item is the highest priority item in the queue or the requested access class if that is the highest priority item in the queue. The caller should check the returned access class.

VIII. TASK MANAGER SPECIFICATION

The Task Manager is responsible for manager the several single level tasks within the process. It implements the following functions:

- Create a new task
- Destroy a task
- Switch a task
- Get work for a task
- Add memory to a task address space

The Task Manager maintains a database of the tasks being managed and the access class of those tasks. Tasks request new transactions from the Task Manager. The Task Manager interfaces with the appropriate MLS queue to retrieve the transactions. If no transactions of the requested access class are available, the current task is suspended and a task for which a transaction does exist is scheduled.

A. TASK MANAGER CONSTANTS

The task database is indexed based upon access class. Therefore, the possible access classes in the system would be converted to an ordered list, indexed starting at 1 for the first access class and going to MAX_ACCESS_CLASS for the last access class in the ordered list. The index assigned to each access class does not denote any relationship or precedence between the access classes, it is merely used for indexing.

a. MAX_ACCESS_CLASS – The maximum number of tasks the Task Manager can handle

B. TASK MANAGER DATABASES

a. Task Database

The task database stores information about the various tasks being managed by the Task Manager. The task database is indexed by access class, therefore, there can be only one task per access class.

Each entry in the task database has the following format

```
struct Task_Entry {
    LDT_Database_Entry : ldt_id;      // the LDT image for this task
    Task_Status : status;             // the status of this task
    Access_Class: access;             // the access class of this task
    Task_Context: context;            // the saved task context
    Task_Entry : next;                // the next task in the list
    Task_Entry : prev;                // the previous task in the list
}
```

The database is an array of task entries

```
Array of Task_Entry : Task_Database[1..MAX_ACCESS_CLASS];
```

C. TASK MANAGER VARIABLES

a. Current_Task – the access class (index) of the current task (this uniquely identifies the current task)

b. MLS_Que – The MLS queue associated with this process (all tasks)

D. TASK MANAGER FUNCTIONS

a. tm_create_task

Function: tm_create_task(Access_Class: access): Success_Code

Purpose: This function initializes a new TP task to process transactions of a given access class.

Inputs:

- Access_Class: access – The access class at which to create the new task

Outputs:

- Success_Code – indicates the result of the operation. Possible values include:
  - SUCCEEDED – The operation completed successfully
  - NO_LDT_SPACE – The function was unable to allocate an LDT
  - TASK_IN_USE – A task of the given access class already exists
  - INVALID_ACCESS_CLASS – The specified access class is not valid
  - FAILED – The operation did not complete successfully

Processing:

```
{
    // Local variables
    Success_Code              return_code;   // success of error code
    LDT_Database_Entry new_ldt;              // new ldt for task
```

```
return_code = SUCCEEDED;
// default case is failure
return_code = FAILED;
// Ensure we have a valid access class
if (valid_access_class(access) ≠ TRUE) {
        // set error code
        return_code = INVALID_ACCESS_CLASS;
// Ensure we don't already have a task at the given access class
if (Task_Database[access] ≠ FREE) {
        // set error code
        return_code = TASK_IN_USE; }
// ensure we can allocate a new LDT for the task
else if (ldt_create_ldt(access, new_ldt) ≠ SUCCEEDED) {
        // set error code
        return_code = NO_LDT_SPACE; }
// otherwise, initialize the task
else {
        // assign the new LDT
        Task_Database[access].ldt_id = new_ldt;
        // mark this task as in use
        Task_Database[access].status = READY;

        // **** Additional setup of the address space and context would
        // **** go here.  Such things as settting up the code segment, initial
        // **** LDT, initial selectors, initial program counter, etc...

}
```

```
    return return_code;
}
```

Effects:

- If successful, a new task at the specified access is created and initialized. The identifier associated with the new task will be the access at which it was created.

b. tm_destroy_task

Function: tm_destroy_task(Access_Class: access): Success_Code

Purpose: This function destroys the task at a given access class.

Inputs:

- Access_Class: access – The identifier (access class) for the task to destroy

Outputs:

- Success_Code – Indicates the result of the operation. Possible values include:
  - SUCCEEDED – The operation completed successfully
  - INVALID_ACCESS_CLASS – The specified access class is not valid
  - NO_TASK – There is no task at the specified access class
  - FAILED – The operation did not complete successfully

Processing:

```
{
    // Local variables
    Success_Code        return_code;   // success or error code

    return_code = SUCCEEDED;
    // Ensure we have a valid access class
    if (valid_access_class(access) ≠ TRUE) {
            // set error code
            return_code = INVALID_ACCESS_CLASS;
    // Ensure we don't already have a task at the given access class
```

```
    if (Task_Database[access].status == FREE) {
            // set error code
            return_code = NO_TASK; }
    // otherwise, remove the task
    else {

            // **** Deallocation of task memory and resources should occur
            // **** here.  The code segment and other segment held might have
            // **** to be deallocated.  Once this is done, we can deallocate
            // **** the LDT

            ldt_destroy_lde(Task_Database[access].ldt_id);
            // mark this task as free
            Task_Database[access].status = FREE;
            return_code = SUCCEEDED;
    }
    return return_code;
}
```

Effects:

- If successful, the task of the specified access class will have its resources allocated and marked as free.

c. tm_switch_task

Function: tm_switch_task(Access_Class: access): Success_Code

Purpose: This function switches to the task corresponding the specified access class

Inputs:

- Access_Class: access – The identifier (access class) of the task we would like to switch to

Outputs:

- Success_Code – Indicates the result of the operation. Possible values include:
  - SUCCEEDED – The operation completed successfully
  - INVALID_ACCESS_CLASS – The specified access class does not exit
  - NO_TASK – There is no task of the specified access class
  - FAILED – The operation did not complete successfully

Processing:

```
{
    // Local variables
    Success_Code        return_code;   // success or error code

    // default case is failure
    return_code = FAILED;
    // Ensure we have a valid access class
    if (valid_access_class(access) ≠ TRUE) {
            // set error code
```

```
            return_code = INVALID_ACCESS_CLASS;
    // Ensure we don't already have a task at the given access class
    if (Task_Database[access].status == FREE) {
            // set error code
            return_code = NO_TASK; }
            // otherwise, make the switch
    else {
            // save the current task
            save_task_context(Task_Database[Current_Task].context);
            Task_Database[Current_Task].status = SUSPENDED;
            // restore the context of the new task
            restore_task_context(Task_Database[access].context);
            // switch the LDT
            ldt_switch_ldt(Task_Database[access].ldt_id);

            // **** Other steps as necessary to save and restore the state
            // **** go here.
            }
    return return_code;
}
```

Effects:

- If successful, the current task will have its context saved and the new task (of the specified access class) will have its context restores and its LDT loaded.

IX. CONCLUSION

There are applications within the military that would benefit from the existence of an MLS TP system. A preliminary three-tier architecture that provides a rudimentary TP system based upon the abstractions of task, process and kernel has been presented. This architecture avoids the normally heavy cost of a context switch in an MLS system by leveraging the security features of the Intel Pentium microprocessors. By effectively using the privilege level and descriptor table mechanism of these processors, the required processing to switch between access classes can be significantly reduced.

What has been presented here is the preliminary architecture for such a system. Implementation, using the presented specification for scaffolding, can be the subject of future work. The addition of a file-system, memory management (to include paging), and device management would greatly add to the functionality of the system.

LIST OF REFERENCES

Anderson, James P., *Computer Security Technology Planning Study*, Report ESD-TR-73-51, U.S. Air Force Electronics Systems Division, Bedford, MA, 1972.

Bell, D. E. and LaPadula, L. J., *Secure Computer System: Unified Exposition and Multics Interpretation*, Report MTR-2997 Rev. 1. AD A023 588, The Mitre Corporation, Bedford, MA, 1976.

Bernstein, P. A. and Newcomer, E., *Principles of Transaction Processing*, Morgan Kaufmann, San Francisco, CA, 1997.

Biba, K. J., *Integrity Considerations for Secure Computer Systems*, Report ESD-TR-76-372, AD A039324, U. S. Air Force Electronics Systems Division, Bedford, MA, 1977.

Brinkley, D. L. and Schell, R. R., "Concepts and Terminology for Computer Security", *Information Security: An Integrated Collection of Essays*, IEEE Computer Society Press, Los Alamitos, CA, 1995, pp. 40-97.

Department of Defense, *Department of Defense Trusted Computer System Evaluation Criteria*, DoD 5200.28-STD, National Computer Security Center, 1985.

Intel Corporation, *Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture*, Intel Corporation, Mt. Prospect, IL, 1997.

Intel Corporation, *Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference*, Intel Corporation, Mt. Prospect, IL, 1997.

Intel Corporation, *Intel Architecture Software Developer's Manual, Volume 3: System Programming Guide*, Intel Corporation, Mt. Prospect, IL, 1997.

Kang, M. H., Froscher, J. N., and Moskowitz, I. S.. "An Architecture for Multilevel Secure Interoperability." In *Proceedings of the 13th Annual Computer Security Applications Conference*. IEEE Computer Society, Los Alamitos, Calif., 1997, pp. 194-204.

Saltzer, J. H. and Schroeder, M. D., "The Protection of Information in Computer Systems", In *Proceedings of the IEEE,* 63, 9, 1975, pp. 1278-1308.

Schroeder, M. D. and Saltzer, J. H, "A Hardware Architecture for Implementing Protection Rings", In *Communications of the ACM*, 15, 3, 1972, pp. 157-170.

Shirley, L. J. and Schell, R. R., "Mechanism Sufficiency Validation by Assignment", In *Proceedings of the IEEE*, IEEE Computer Society Press, Oakland, CA, 1981, pp. 26-32.

Shockley, W. R. and Schell, R. R., "TCB Subsets for Incremental Evaluation", In *Proceedings of the Third AIAA Conference on Computer Security*, 1987, pp. 131-139.

Stallings, William, *Operating Systems: Internals and Design Principles*, Third Edition, Prentice-Hall, Upper Saddle River, NJ, 1998.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center..... 2
   8725 John J. Kingman Road, Ste. 0944
   Ft. Belvoir, Virginia 22060-6218

2. Dudley Knox Library..... 2
   Naval Postgraduate School
   411 Dyer Rd.
   Monterey, CA 93943-5101

3. Dr. Dan Boger..... 1
   Chairman, Code CS
   Department of Computer Science
   Naval Postgraduate School
   Monterey, CA 93943-5121

4. Dr. Cynthia E. Irvine, Code CS/Ic..... 3
   Department of Computer Science
   Naval Postgraduate School
   Monterey, CA 93943-5121

5. Mr. William R. Shockley..... 1
   Cyberscape Computer Services
   1885 Franklin Street
   Lebanon, OR 97355

6. CAPT Dan Galik..... 1
   Space and Naval Warfare Systems Command
   PMW 161
   Building OT-1, Room 1024
   4301 Pacific Highway
   San Diego, CA 92110-3127

7. Commander, Naval Security Group Command..... 1
Naval Security Group Headquarters
9800 Savage Road
Suite 6585
Fort Meade, MD 20755-6585
ATTN: Mr. James Shearer

8. Mr. George Bieber..... 1
Defense Information Systems Agency
Center for Information Systems Security
5113 Leesburg Pike, Suite 400
Falls Church, VA 22041-3230

9. Joseph O'Kane..... 1
National Security Agency
Research and Development Building
R23
9800 Savage Road
Fort Meade, MD 20755-6000

10. CDR Chris Perry..... 1
N643
Presidential Tower 1
2511 South Jeffersion Davis Highway
Arlington, VA 22202

11. LT Haruna R. Isa..... 1
2112 Cooper Avenue
Sheboygan, WI 53083